

GNO/ME Version 2.0

By Jawaid Bazyar and Tim Meekins

The GNO Multitasking Environment is Copyright 1991-1993 by Procyon Enterprises Incorporated

The ORCA/C run-time libraries are Copyright 1987-1993 Byte Works, Inc.
and distributed with permission.

This product includes software developed by the University of California, Berkeley and its contributors.

GNO/ME 2.0 also includes several utilities and libraries produced by outside authors and in the public domain. This software is included solely as a convenience to users of GNO/ME, and is not considered part of GNO/ME for copyright purposes.

GNO and GNO/ME are trademarks of Procyon Enterprises Incorporated.

Apple IIGS, APW, Finder, GS/OS, ProDOS, Macintosh, and HFS are registered trademarks of Apple Computer, Inc.

AppleWorks GS is a trademark of Claris Corp.

UNIX is a registered trademark of AT&T Bell Laboratories.

Important Notice: This is a fully copyrighted work and as such is protected under the copyright laws of the United States of America. According to these laws, consumers of copywritten material may make copies for their personal use only. Duplication for any other purpose whatsoever would constitute infringement of copyright laws and the offender would be liable to civil damages of up to \$50,000 in addition to actual damages, plus criminal penalties of up to one year imprisonment and/or a \$10,000 fine.

Procyon Enterprises Inc. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED IN SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

This product is sold for use on a *single computer* at a *single location*. For information on obtaining a site license for using multiple copies, contact the publisher.

Procyon Enterprises, Inc.

P.O. Box 620334

Littleton, CO 80162-0334 USA

(303) 781-3273

Credits

"Always listen to experts. They'll tell you what can't be done, and why. Then do it."
Lazarus Long

The **GNO Multitasking Environment** for the Apple IIgs
Copyright 1991-1993, Procyon Enterprises Inc. and Tim Meekins

Please direct all inquiries to:

Procyon, Inc.
PO Box 620334
Littleton, CO 80162-0334 USA
(303) 781-3273

For on-line technical assistance, contact:

America OnLine	: GNOJawaid, GNOTim2
GEnie	: Procyon.Inc
Internet	: bazyar@cs.uiuc.edu
Delphi	: JAWAIDB

Written by:

Jawaid Bazyar	Kernel & Documentation
Tim Meekins	Shell & Documentation
Albert Chin	Documentation
Andrew Roughan	Documentation
Derek Taubert	Kernel support and utilities
Greg Thompson, Philip Vandry,	
James Brookes, Ian Schmidt	Utilities

With many thanks to:

Bill Gulstad, Rob Knauerhase, and everyone on the GNOBETA Internet mailing list
Dave Lyons, for tolerating my endless questions since the beginning of time, for getting me started in IIgs programming, and for calling GNO 'cool'.
Mike Westerfield, for your technical assistance and all your work in making great development tools for the IIgs.

Special thanks to:

Matt Deatherage, for egging me on to complete the project; also, for your most helpful technical support through DEVSUPPORT- you guys made this thing possible!
Randy Hyde, for telling me it couldn't be done.

MISCELLANEOUS

Reporting Bugs

In any large piece of computer software such as the GNO/ME system, bugs are sure to turn up, no matter how much testing is performed on the software before it goes out the door. If you discover a bug in GNO/ME, we'd like to hear about it. There are several things we require, however, to make bug reports useful to us.

First of all, we need a complete description of your computer system; how much RAM, what cards, in what slots, what type of disk storage, etc. This information is very important in tracking down hardware-dependent bugs. Also, we need to know the version numbers of the software involved; the GNO Kernel, the GNO Shell, and any utilities.

Second, we need a step-by-step description of how to duplicate the bug. If this requires writing down individual keystrokes, then we need it. Only in this way can we decide whether the bug is hardware dependent or global.

You can send bug reports to any of the electronic mail addresses listed on the 'Credits' page, or by mailing a disk containing a description of the problem and the necessary software and files to the Procyon address.

User Projects

If you're working on a project which utilizes GNO/ME in some way, we'd like to know about it. Just contact us by mail, phone, or whatever's most convenient for you, and tell us about your project. If possible, we'll coordinate your efforts with those of other programmers. If we really like your project and think it may be useful to others, we may include it on the next GNO/ME distribution!

Software Piracy

If you copied GNO/ME from someone and are now reading this, take a moment to reflect on what you've done. GNO/ME is a project that has consumed around three man-years of effort, huge amounts of money, and a lot of grief on our part. The kernel alone consists of over 18,000 lines of source code; the shell and utilities make up another twenty to thirty thousand lines.

Is it right that you're now benefitting from our effort without any just compensation to the authors and Procyon, Inc.? We are Apple II programmers born, bred and raised in the spirit of the machine; we love to program it, and probably always will. But if we cannot make enough money to make it worthwhile to continue work on GNO/ME, then we won't; because by worthwhile, we mean not only food on the table, but resources for expansion; expansion of our IIgs product line and support to IIgs owners being abandoned by the rest of the industry. In short, support us and we'll support you.

GNO/ME is very inexpensive when you consider what it allows you to do, and what it will allow you to do when software development for GNO/ME starts full-swing. And it's a small price to pay indeed for our loyalty to you, the IIgs owner.

To all you GNO/ME purchasers out there: thank you very much. We will continue to provide you with new IIgs products until our hair and teeth fall out, and our fingers are too frail to type the keys (and even then, with Easy Access, we might crank out a program or two). We hope you will be sufficiently impressed with the system to recommend it to others.

p.s. Remember - if someone comes up to you on the street and offers you some IBM, **Just Say GNO!**

PREFACE

Computers are tools. The flexibility of a tool determines how useful it is. Early computers were much like the one this software was written for: the Apple IIgs. They could only run one program at a time, and their usefulness was limited to what the particular program the user was executing offered. In the late 1960's, a team of researchers at AT&T began developing the UNIX operating system. The UNIX design was partially based on the premise that most programs are I/O bound, that is, most of the time the program executes is spent waiting for user input or other I/O operations. While one program is waiting for I/O, why not allow another program to execute? This is what they did, and the result was one of the most successful computer operating systems ever created.

The Apple IIgs, like the Macintosh it is modelled after, provides very limited multitasking abilities in the form of desk accessories (NDAs). The programs in the NDA menu are available in whatever application you use as long as it follows Apple's guidelines. However, there are many graphics based programs that don't support NDAs, and in addition there is a wealth of software that has been developed for the Byte Works' ORCA environment. This environment is mainly text-based, and thus makes access to NDAs impossible. As if that wasn't enough, it's very difficult to write an NDA to allow the application to keep running concurrently. So the benefits are lost, and we're back at ground zero.

Enter the GNO Multitasking Environment. What was once just dreamed about is now a reality. GNO/ME provides an environment that is almost entirely compatible with software developed for the ORCA environment. But GNO/ME also provides a wealth of new abilities, lots of new ground for developers and users alike.

Before we begin describing, we'd like to respond to those who say such a multitasking system isn't possible on the Apple IIgs. Obviously it is: you hold it in your hands. Some say the Apple IIgs isn't powerful enough to make multitasking useful. We point out that the Apple IIgs is much more powerful than the first computers UNIX was designed to run on; they only had 64K of real memory, and were 16 bit machines. Some ask why you'd ever need to run more than one program at once. These are the same people who asked why we'd ever need more than 64K of memory, or more than 140K of storage on disks (end soapbox).

Introduction

The GNO Multitasking Environment provides pre-emptive multitasking. Many programs can be executing at the same time; each is called a 'process'. Each process is allowed to run for a short period of time (1/20th of a second on average). When its time runs out, the current process is set aside and another one chosen to run next. This cycle continues until there are no more processes left (i.e. when you exit GNO/ME). Starting up processes to run 'in the background' is a simple matter of adding a few characters to the shell commands.

GNO/ME provides a shell that takes full advantage of the multitasking ability provided. The most important feature of the shell is job control (starting, terminating, and suspending processes). But the shell also provides power never before seen on the Apple IIgs. The ability to choose files by 'wildcard' has been around for a while, but the GNO Shell takes this to a new level with 'regular expressions', a very powerful yet simple programming language. Other benefits of the GNO shell are too numerous to mention. (see the *GNO Shell User's Manual* for details).

In addition to being compatible with the ORCA system, GNO/ME is a very powerful programming environment. Available to the programmer are all the calls needed to control processes, support Inter-Process Communication, and other tools needed in a multitasking environment.

GNO/ME also boasts the first completely consistent method for accessing serial and console I/O. The IIgs TextTools have been incredibly enhanced to provide a truly all-encompassing interface for serial, console, and IPC applications. Imagine being able to attach terminals to your GS, and have a useful shell in each one. Multiuser BBSs, remote dial-ups, UUCP or SLIP that doesn't take over your computer- the applications are endless!

With all this talk of shell utilities, have desktop users (users of programs like AppleWorks GS) been left behind? Absolutely not. GNO/ME doesn't allow more than one desktop program to run concurrently, but it DOES let you run a desktop program with as many text applications as you like. In other words, no functionality is lost from the IIgs by using GNO/ME.

Finally, the GNO Multitasking Environment comes with a large number of free utilities that bring some of the power of a UNIX system to the Apple IIgs. Also, a number of programming libraries are included that make it easy to port programs from UNIX or MS-DOS systems to the Apple IIgs.

The GNO/ME package

Included in your GNO/ME Version 2.0 package are:

- this GNO/ME overview
- the GNO Shell User's Manual
- the GNO Kernel Reference Manual
- a selection of utility and library documentation
- a reading list containing a wide selection of books for both the user and the programmer
- three disks containing the GNO Kernel, GNO Shell, and loads of utilities

Hardware Requirements

GNO/ME will work on any Apple IIgs with at least 2 MegaBytes of memory and a hard drive. You should have at least 5 MegaBytes of hard disk space free.

We recommend 4 MegaBytes of Memory and an accellerator card, especially if you will be using GNO with many background processes. A modem and access to an on-line service will greatly speed access to technical assistance and new utilities as they are made available.

Installation

GNO/ME is a very large system. Most of the system is stored on the three disks in a compressed format. For this reason we recommend that you read this section carefully and use the included automated installation program to save yourself the major headache of arranging everything. You'll need a hard drive partition with at least 5 Megabytes free to install GNO/ME. The entire installation process should take no longer than 10 minutes.

If you are upgrading from GNO 1.0 to GNO 2.0, we recommend that you install GNO 2.0 on a new area of your hard drive. This is because GNO 2.0 provides new versions of almost all the utilities and other software provided with GNO 1.0. You can then move the parts of your old system that aren't a part of the standard GNO 2.0 distribution over to the new installation.

The installer program is able to copy the GNO libraries for use with the ORCA languages and tools. If you want to use your ORCA languages and tools, you should answer the appropriate questions during the install process. ORCA installation is covered in more detail below.

Step 1:

Copy the contents of /GNO.Disk1 to the partition where you want to install GNO/ME. The easiest way to do this is to run the IIGS Finder™ program, insert the /GNO.Disk1 disk, and drag the /GNO.Disk1 onto the icon of the drive you want to install on. The Finder will tell you that the disks are of a different size, and ask you if you wish to place the contents of /GNO.Disk1 in a folder on the hard drive partition. Answer "Yes", and the Finder will copy the data.

Step 2:

Open the newly created GNO.Disk1 folder on the partition and execute 'Kern' by double-clicking on the Kern icon. GNO.Disk1 contains a minimal GNO system which you have just started up. You'll see some copyright messages, and after a few seconds the desktop will appear.

Step 3:

Install will ask you questions before the installation process begins. Questions asked by Install are boldfaced in the following discussion. If you're uncertain of how to answer, click on the 'Info' button to get more information on the question. This information is reproduced here for ease of reference.

Do you have existing Byteworks (ORCA) language products?

If you use Byteworks language products such as ORCA/C, ORCA/M, ORCA/Pascal, etc. and wish to use them from GNO/ME then answer 'Yes' to this question. Otherwise, answer 'No'.

Clicking on the 'Info' button will display the following text:

If you currently use products such as ORCA/C, ORCA/Pascal, ORCA/M, etc., and wish to use them from GNO/ME then you should answer 'Yes' to this dialog.

Install will then ask you to locate the file "ORCA.SYS16" (the ORCA Shell). Usually, it will be in a directory called "ORCA" on one of your hard drive partitions. On our system, it is :software:orca. The location of this file determines where Install puts the GNO-specific ORCA/C header files and libraries (in the ORCA LIBRARIES directory). If you have renamed ORCA.SYS16 to something else, select that instead.

Install uses this information to correctly set the ORCA prefixes and add the EXECUTABLES directory to the \$PATH variable in your gshrc file. See the GNO Shell User's Manual, Appendix B, for more information on the ORCA prefixes.

Do you want to install the modified ORCA/C 2.0 ORCALIB library?

If you have ORCA/C 2.0 installed and you answered 'Yes' to the previous question, then you should answer 'Yes' now. If you have ORCA/C 1.3 installed, or you do not have any ORCA languages or tools, then you should answer 'No'.

Clicking 'Info' will display the following text:

GNO/ME 2.0 comes with a modified version of the ORCA/C ORCALIB standard library. This library adds specific support for some of GNO's special files (pipes, terminals, etc) transparently and corrects other problems the standard ORCALIB has with GNO.

This special ORCALIB is only for use with ORCA/C 2.0. It will NOT work with previous versions of ORCA/C.

Note that all of GNO's other libraries (libgno, libbsd, libc) work fine with ORCA/C 1.3.

Do you want to install the Multi-User package?

If you wish to have the Multi User package installed, answer 'Yes'. It is recommended that new users do not install this package until they are familiar with the system. This package is not fully discussed in the GNO/ME manuals and may introduce many unfamiliar concepts.

Clicking 'Info' will display the following text:

The 'Multi-User' package consists of a number of cooperating programs, including init(8) and login(8). MU provides for password-protected access to the GNO Shell via the console or serial ports (modems and dedicated terminals). It also handles multiple user accounts, and a certain level of protection between users.

If you will be using a terminal with GNO, or want to be able to dial into GNO with a modem, you should install MU.

You may wish to hold off setting up the Multi-User package until you learn your way around GNO a little better, as it's simple to set up MU later.

Step 4:

The Installer will now proceed to arrange the GNO Multitasking Environment on your hard drive. Initially, the contents of /GNO.Disk1 will be moved into the proper places on the volume you have chosen. When asked by the Installer, you should place /GNO.Disk2 and /GNO.Disk3 into the disk drive. The contents of these disks will be extracted and decompressed into their proper places.

If the installer cannot finish it's tasks for some reason, it will abort to the shell. At this point you should study the error messages to determine the cause of the failure,

Step 5:

When the Installer finishes, you will be returned to the GNO prompt. At this time you should type 'exit' to return to Finder and delete the GNO.Disk1 directory, as it is no longer needed (GNO is completely installed in a subdirectory 'GNO' on the same partition GNO.Disk1 is on).

That's all!

What has been installed?

If you examine the directory that you chose for GNO/ME, you will find that the Installer has created several directories. Here is a brief rundown of thier contents:

:gno:bin:	This directory contains the executable shell utilities included with the GNO/ME package.
:gno:dev:	This directory contains device drivers. The GNO/ME package includes serial drivers for the IIGS' built-in modem and printer ports.
:gno:etc:	Various system configuration files
:gno:initrc	This a startup file for kern and must remain in this directory.
:gno:kern	This is the GNO kernel. Launch this to start GNO/ME.
:gno:lib:	This directory is provided for,,,, It is currently empty.
:gno:tmp:	This directory is provided for,,,, It is currently empty.
:gno:user:	This directory is created if you installed the Multi-User package. It contains a user directories for users of the system. Currently there is only one: 'root'.
:gno:usr:	This directory contains UNIX System Resources (USR).
:gno:usr:bin:	Miscellaneous utilities
:gno:usr:man:	This directory contains the manual pages for the 'man' utility. There are eight categories of information as defined by UNIX standards. These eight categories are presented in either formatted (man) or unformatted (cat) entries.
:gno:usr:games:	The classic "Hunt the Wumpus" game, and perhaps others.
:gno:usr:sbin:	System administration utilities.

Desk Accessories

The following is a description of the desk accessories that are shipped with GNO/ME.

GSI NDA

The Graphical Shell Interface New Desk Accessory allows you to access the shell from a window inside a desktop application. Selecting 'GSI' from the Apple menu opens a window and brings up a shell in it. You can use the shell in this window just as you would in text mode.

At this time, GSI doesn't support any terminal emulations; i.e., it's a 'dumb' interface. As a result, full screen programs like 'less' and 'vi' will not work properly in GSI.

Suspend NDA

The Suspend New Desk Accessory allows you to temporarily stop a desktop program and return to the text shell. To return to the desktop program, simply type 'fg'.

You may put desktop programs in the background ('bg') at your own risk. This is not guaranteed to work for all programs, and putting a desktop program in the background will not put the Apple IIgs in text mode.

Also, do not run another desktop program when one is currently suspended. The GNO kernel tries to cleanly terminate the second program, but may not always be able to accomplish this.

GNO Snooper CDA

The GNO Snooper CDA is a utility that is very helpful in debugging software, and reporting system crashes. Snooper has access to information inside the kernel that is not generally available to processes.

When you enter the control panel and choose "GNO Snooper" a process list is displayed, which is very similar to the 'ps' shell command, and a menu is presented with the following options:

D)etails

Get detailed information on a process. Snooper asks you for the process ID number of the process you want to investigate. Snooper then dumps the state of the 65816 registers and shows various other information about the process.

K)ill

This option sends a SIGKILL (signal number 9) to a process. Just type in the process ID of the process you want to kill.

P)rocess Group Dump

Dumps the system process group tables, which is very useful for debugging code which manipulates process groups. The first table, 'pgrp', is the process group reference count table. This table lists how many processes (and TTYs) belong to that process group. pgrp ID's start at 2, so the first pgrp listed is 2.

The second listing is the ttyStruct table. This table shows which pgrp each TTY belongs to. The TTYs start with .null (0), then move on to .ttyb, .ttya, .ttyco, etc.

F)ile Table Dump

Lists all open files that GNO knows about. The information is presented in tuples. The first element is the real GS/OS refNum or device driver ID. The second is the type of file (GSOS, ttyXX, or PIPE). The third is how many references to that file are open. This information is global, so if two processes have .NULL opened, the refcount for .tty00 will be 2.

Reading List

Recommended Reading for Users

Title: Learning Unix
Author: James Gardner
Publisher: Sams
Edition: 1991
ISBN: 0-672-30001-X

Comment: With disks containing MSDOS simulation of Unix (MKS Tools). A good tutorial / reference book for those without constant access to Unix.

Title: The Unix Operating System
Author: Kaare Christian
Publisher: Wiley

Edition: 2nd ed. 1988

ISBN: 0-471-84781-X

Comment: A classic overview of Unix commands. Good in coverage.

Title: Peter Norton's Guide to Unix
Authors: Peter Norton and Harley Hahn
Publisher: Bantam Computer
Edition: 1991

ISBN: 0-553-35260-1

Comment: Good coverage. A good introduction for beginners (especially those accustomed to DOS).

Title: Unix in a Nutshell
Authors: Daniel Gilly and O'Reilly staff
Publisher: O'Reilly
Edition: 2nd ed. 1992 (System V and Solaris 2)
ISBN: 1-56592-001-5

Comment: **** Highly Recommended **** An excellent desktop reference to almost all Unix commands "a complete reference containing all commands and options, plus generous descriptions and examples that put the commands in context." Also, an edition for 4.3. BSD.

Title: Life with Unix - A Guide for Everyone
Authors: Don Libes and Sandy Ressler
Publisher: Prentice Hall
Edition: 1990
ISBN: 0-13-536657-7

Comment: **** Highly Recommended **** An everything-you-want-to-know-about-Unix book. It includes info you might not find elsewhere. "This book is the "other" book about Unix, a study in reading between the lines - which is very much what learning UNIX is like."

Title: Unix for the Impatient
Authors: Paul Abrahams and Bruce Larson
Publisher: Addison Wesley
Edition: 1992

ISBN: 0-201-55703-7

Comment: **** Highly Recommended **** A new, comprehensive, in-depth reference to Unix. "a handbook you can use both as a manual to learn UNIX and as a ready reference for fast answers to specific UNIX questions."

Title: Learning the vi Editor

Author: Linda Lamb

Publisher: O'Reilly

Edition: 1990

ISBN: 0-937175-67-6

Comment: A very good guide to vi and ex commands. With a quick reference card.

Titles: vi Tutor and vi Reference

Authors: Micheal Pierce and Robert Ware (Tut), Maarten Litmaati (Ref)

Edition: 1.3 (Tut), 8 (Ref)

Comment: These and other good vi stuff are obtainable by anonymous ftp from cs.uwp.edu (in pub/vi) ...

Recommended Reading for Programmers

Title: The Design and Implementation of the 4.3 BSD Unix Operating System
Authors: Samuel Leffler et al
Publisher: Addison-Wesley
Edition: 1990

ISBN: 0-201-06196-1

Comment: An authoritative description of the design of BSD Unix. "It covers the internal structure of the 4.3BSD system and the concepts, data structures, and algorithms used in implementing the system facilities."

Title: The Unix Programming Environment
Authors: Brian Kernighan and Rob Pike
Publisher: Prentice-Hall
Edition: 1984

ISBN: 0-13-937681-X

Comment: A true classic on Unix programming.

Title: Advanced Programming in The Unix Environment

Author: Richard Stevens

Publisher: Addison-Wesley

Edition: 1992

ISBN: 0-201-56317-7

Comment: **** Highly Recommended **** A going-to-be classic on how programs work under Unix. The source codes and errata list are obtainable by anonymous ftp from ftp.uu.net (in /published/books).

Title: Advanced Unix Programming

Author: Marc Rochkind

Publisher: Prentice Hall

Edition: 1985

ISBN: 0-13-011818-4

Comment: A superb book covering all system calls in detail.

Title: Using C on the Unix System

Author: David Curry

Publisher: O'Reilly

Edition: 1990

ISBN: 0-937175-23-4

Comment: This book is directed to (would-be) system programmers.

Required Reading for Programmers

Title: The C Programming Language

Authors: Brian Kernighan and Dennis Ritchie

Publisher: Prentice Hall

Edition: 2nd ed. 1988

ISBN: 0-13-110362-8

Comment: The answers to the exercises can be found in C Answer Book written by Tondo and Gimpel and published by Prentice Hall (ISBN: 0-13-109653-2).

All the books listed below are available from:

Resource Central

PO Box 11250

Overland Park Kansas 66207

(913) 469 6502

Title: Apple IIs ToolBox Reference.

Volumes 1, 2 & 3

Author: Apple Computer

Publisher: Addison-Wesley

Title: Programmers Reference for System 6.0

Author: Mike Westerfield

Publisher: Byteworks, Inc

Title: Apple IIs Firmware Reference

Author: Apple Computer

Publisher: Addison-Wesley

Title: GS/OS Reference

Author: Apple Computer

Publisher: Addison-Wesley

GNO Shell User's Manual

By Tim Meekins, Albert Chin, and Jawaid Bazyar
Edited by Andrew Roughan

Table of Contents

Chapter 1	Getting started with the GNO Shell	3
	Introduction.....	3
	Customizing the Shell Environment.....	3
	Invoking gsh.....	5
Chapter 2	Interacting with the GNO Shell	7
	Executing Commands.....	7
	Command-line Editor.....	7
	Command Input.....	8
	Command Editing.....	8
	History Editing Keys.....	9
	Command, Filename and Variable completion.....	10
Chapter 3	Using the GNO Shell more productively	13
	What does this command do?.....	13
	Option Arguments.....	13
	Entering Multiple commands.....	15
	Using Aliases as Shorthand.....	15
	Redirecting Input and Output.....	16
	Pipelines.....	16
	Background Execution of Commands.....	17
	Job Control.....	19
	Working with Pathnames.....	21
	Pathname Expansion.....	22
	Quoting Special Characters.....	23
	How gsh Finds a Command.....	24
Chapter 4	Built-in Commands	25
	Built-ins vs EXE Commands.....	25
	Shell Commands.....	25
	Kernel Commands.....	29
	Environment Commands.....	30
Chapter 5	Shell Variables	33
	Using shell variables.....	33
	Scope.....	33
	Description of Pre-defined Shell Variables.....	33
	Accessing shell variables.....	35
Appendix A	Sample gsh session	39
Appendix B	Prefix Conventions	45
Appendix C	Errors	46
Appendix D	Signals	49
Appendix E	Non-Compliant Applications	50
Appendix F	Termcaps	51
Glossary		53
Index		55

Chapter 1

Getting started with the GNO Shell

"Computer operating systems are among the most complex objects created by mankind..."
Douglas Comer, Operating System Design, The Xinu Approach

Introduction

The GNO shell is an integral part of the GNO Multitasking Environment (GNO/ME). The GNO shell provides the interface between the user and the GNO Kernel. While both work together, the jobs they perform are quite different. This manual documents the functions of the shell.

The user interacts with the shell through a command-line interface. Command-line interfaces provide a unique way of interacting with the operating system. Unlike GUIs (Graphical User Interfaces), with which you are already familiar with by using programs such as the Finder and ShrinkIt! GS, all commands are typically entered using the keyboard. The shell interprets commands and passes them to the kernel for control and execution.

The command-line interface will be unfamiliar to some people. However, once the command-line interface has been mastered, the user should have no difficulty using any current or future GNO applications. Those of you already familiar with Unix interfaces, such as the C shell, Bourne shell, and Korn shell, or the ORCA shell on the Apple IIGS, will begin to realize the advantages which GNO/ME is able to provide.

The way this manual is presented allows the complete beginner to simply work through the chapters in a chronological order. Chapter 2 familiarises the user with entering basic commands whereas the more powerful GNO/ME features are introduced in Chapter 3. Chapter 4 documents the commands which are built into the GNO Shell and Chapter 5 explains shell variables which give the user control over how their installation functions.

Customizing the Shell Environment

When **gsh**, the implementation of the GNO Shell, is executed, it reads in and processes the *gshrc* file. This file contains start-up instructions for the shell, which can be used to customize the operation of the shell and other aspects of the system. It is created by the GNO Installer during the installation process.

The following is a sample *gshrc* file (line numbers have been added for convenience):

###	line #1
#	line #2
# GNO 2.0 gshrc file	line #3
#	line #4
###	line #5
#	line #6
# Initialize our environment	line #7
#	line #8
set path=":hard:gno:bin :hard:gno:usr:bin"	line #9
set prompt="[%h] %S%t%s %C> "	line #10
set home=":hard:gno:user:root"	line #11
set term=gnocon	line #12
export path prompt home term	line #13
setenv history=100 savehist=25	line #14


```

### line #15
# line #16
#Set up standard prefixes for utilities. line #17
# line #18
### line #19
prefix 2 :software:orca:libraries line #20
prefix 3 :software:orca line #21
prefix 4 :software:orca:shell line #22
prefix 5 :software:orca:languages line #23
prefix 6 :software:orca:utilities line #24
prefix 7 :tmp line #25
### line #26
# line #27
# Set up prefixes for Orca2.0(tm)'s benefit line #28
# line #29
### line #30
prefix 13 :software:orca:libraries line #31
prefix 14 :software:orca line #32
prefix 15 :software:orca:shell line #33
prefix 16 :software:orca:languages line #34
prefix 17 :software:orca:utilities line #35
alias ls 'ls -CF' line #36
alias dir 'ls -al' line #37
alias cp 'cp -i' line #38
alias rm 'cp -p rm' line #39
alias mv 'cp -p mv' line #40
setenv usrman='/usr/man' line #41
set fignore='.a .root .sym' line #42
alias zcat 'compress -cd' line #43
setenv pager=less line #44
setenv less=-e line #45
set nonewline=1 line #46
# line #47
# Move to home directory line #48
# line #49
cd line #50

```

When you install GNO/ME, the GNO installer knows where to find the GNO utilities and any ORCA utilities you may have. Unfortunately it does not know where all the other utilities and applications that you may wish to use are located. It is therefore necessary to edit the setup file in order to tell the GNO shell where these programs are on your hard disk.

The setup file, *gshrc* is located in the /usr directory of the path where you installed GNO/ME. You can use any text editor from the desktop to edit the *gshrc* file, or if you are already familiar with the editor *vi* you can use this utility after launching the GNO kernel.

Line 9 is the statement that we are concerned with. **Hard** represents the name of your particular hard drive volume where you have installed GNO/ME.

```
set path=":hard:gno:bin :hard:gno:usr:bin" line #9
```

You will see that spaces have been inserted between pathnames. The space is the pathname separator and the colon has been used as the path delimiter for this specific variable, *PATH*. As an exercise, add your system directory to this statement. Line 9 should now look like this:

```
set path=":hard:gno:bin :hard:gno:usr:bin :hard:system"
```


What you have just done allows the GNO shell to find the **Finder** application. Now go ahead and add any pathnames that hold utilities or applications that you will use frequently from GNO/ME. It should also be noted that it is possible to have more than one pathname containing EXE, SYS16, or EXEC files; this is impossible under ORCA. The PATH variable is discussed thoroughly in Chapter 5.

For now, the remaining lines of the *gshrc* file do not need editing. As you gain an understanding of the system you may wish to make further changes to the *gshrc* file. Make sure you save the file before you exit the editor.

It is possible to modify these instructions while the GNO shell is active, but any changes will be lost upon exiting **gsh**. If you wish the changes to remain effective for the next session you must add them to the *gshrc* file.

By customizing the *gshrc* file it is possible to make the GNO environment more like UNIX, the ORCA environment, or something completely different. Customization of the GNO environment leads to greater user productivity.

Invoking gsh

GNO/ME can be launched from a program launcher, such as the System 6.0 Finder. Launch the GNO Kernel program, **kern** by double clicking on it. The GNO kernel automatically executes the supplied GNO shell, **gsh**.

The prompt, "[1] 2:00am root %", indicates that **gsh** is ready to receive input from the keyboard.

To start a new **gsh** from the command-line simply type **gsh**. If multiple copies of the **gsh** process are undesirable, use the command **source gsh** instead. This is useful for testing changes made to the *gshrc* file. **Source** is a built-in command which is discussed in Chapter 4 **Shell commands**.

Chapter 2

Interacting with the GNO Shell

Executing Commands

A command consists of two parts: a name and its arguments. The command name is the name used to start the command. The name is usually the name of a file which can be executed. The only exceptions are commands which are built-in to the shell. These commands are documented in Chapter 4 **Built-In Commands**. Any shell utility command with a filetype of EXE, SYS16, or EXEC can be executed in this fashion. The command name must be separated from the command arguments with a space.

The command arguments are parameters that the command takes as data to work with (In Applesoft BASIC, "HELLO WORLD" would be an argument for the PRINT command). Command arguments are separated from each other with a space. Note that although arguments extend the usefulness of a command, not all commands have arguments. Any arguments entered after the command will be passed by the shell to the program when it starts executing.

The examples below use the following commands:

<code>qtime</code>	displays time in English text.
<code>echo</code>	prints arguments to the screen.

Examples:

```
% qtime
It's almost five.
% echo II Infinitum
II Infinitum
```

At the simplest level the user enters commands to the shell by typing them on the keyboard. Gsh includes a command-line editor to help the user enter and edit commands. The editor also provides a way to modify and execute previous commands. Additionally the editor can help complete the names of commands, filenames and variables.

Command-line Editor

Below is a complete description of the functions of the command-line editor with short examples depicting how each editing key works.

Throughout the examples the underline character, "_", will be used to represent the current cursor position. In addition, "OA" is used to represent the Open Apple key and the term *word* is used to indicate a string of characters consisting of only letters, digits, and underscores. To the right of a editing key entry is the `bindkey` function name which is used to remap editing functions to new keys. This information is included for reference purposes only. See Chapter 4 **Shell Commands** for more information on the `bindkey` command.

It should be pointed out that at this stage that the user should not be concerned with what the actual commands used in the examples do, rather the user should concentrate on how the command-line editor functions work.

Command Input

These command-line editor keys deal with entering text directly on the command-line.

RETURN	newline The return key is used to terminate line input. gsh then interprets the command on the line and acts accordingly. The position of the cursor on the command-line does not matter. Before <pre>% echo foo bar (RETURN)</pre> After <pre>foo bar % _</pre>
CTRL-D	<no bindkey name> Exits gsh if it was the first character typed on the command-line. If there are still jobs running in the background or stopped, gsh will display "There are stopped jobs." If you press CTRL-D a second time without an intervening command, gsh will terminate all the jobs in the job list and exit.
CTRL-R	redraw Moves to the next line and re-displays the current command-line. Use this to redraw the current line if the screen becomes garbled.
CTRL-L	clear-screen Clears the screen, moves the cursor to the top line, and redraws the prompt and any command-line that was in the process of being edited.

Command Editing

These command-line editor keys allow editing of the command-line text.

CTRL-B LEFT-ARROW	backward-char Moves the cursor one character to the left. You cannot move past the first character on the line. If so, gsh will output an error beep. Before <pre>% print foo bar (LEFT-ARROW)</pre> After <pre>% print foo bar</pre>
CTRL-F RIGHT-ARROW	forward-char Moves the cursor one character to the right. You cannot move past the last character on the line. If so, gsh will output an error beep. Before <pre>% print foo bar (RIGHT-ARROW)</pre> After <pre>% print foo bar</pre>
DELETE	backward-delete-char Deletes the character to the left of the cursor. You can delete up to the first character on the command-line. Before <pre>% print foo bar (DELETE)</pre> After <pre>% print oo bar</pre>
CLEAR CTRL-X	kill-whole-line Deletes all characters on the command line and positions the cursor after the prompt. Before <pre>% print foo bar (CTRL-X)</pre> After <pre>% _</pre>

CTRL-Y	kill-end-of-line Deletes all characters from the cursor to the end of the command-line.
	Before % print <u>foo</u> bar (CTRL-Y)
	After % print f_
CTRL-D OA-D	delete-char Deletes the character under the cursor.
	Before % print <u>foo</u> bar (OA-D)
	After % print fo_bar
CTRL-A OA-<	beginning-of-line Moves the cursor to the beginning of the line.
	Before % print <u>foo</u> bar (OA-<)
	After % print foo bar
CTRL-E OA->	end-of-line Moves the cursor to the first position past the last character on the line.
	Before % print <u>foo</u> bar (OA->)
	After % print foo bar_
OA-RIGHT-ARROW	forward-word Moves the cursor right to the last character of the current word.
	Before % print <u>foo</u> bar (OA-RIGHT-ARROW)
	After % print foo bar
OA-LEFT ARROW	backward-word Moves the cursor left to the beginning of the current word.
	Before % print <u>foo</u> bar (OA-LEFT ARROW)
	After % print <u>foo</u> bar
OA-E	toggle-cursor Toggles input mode between insert and overstrike. Overstrike mode is distinguished by a solid inverse cursor and insert mode by a blinking '_' cursor. In overstrike mode, any characters that are typed directly over-write those characters below the cursor. In insert mode, the characters typed are inserted before the character below the cursor.

History Editing Keys

These command-line editor keys allow access to previously entered commands. The GNO shell automatically keeps track of previous commands in what is called a history buffer.

The maximum number of command-lines saved in the history buffer is determined by the shell variable `HISTORY`. A default value for this variable is set in the `gshrc` file that the GNO Installer creates. The lines saved to the history buffer are kept between sessions. That is, when you exit `gsh`, `$SAVEHIST` command-lines are saved to the `history` file in your `$HOME` directory. When `gsh` is invoked again, all command-lines saved in the history buffer will be available using history editing keys. See Chapter 5 **Description of Predefined Shell Variables** for more information on the `HISTORY` and `SAVEHIST` variables.

CTRL-P **up-history**

UP-ARROW Fetches the previous command-line. If the current command-line is the first line in the history buffer, the next line fetched will be an empty command-line. If invoked again, the last line in the history buffer will be displayed.

Before	After
% print foo bar	
% echo Apple II	
% echo GNO/ME	
% _ (UP-ARROW)	% echo GNO/ME_
% echo GNO/ME_ (UP-ARROW)	% echo Apple II_

CTRL-N **down-history**

DOWN-ARROW Fetches the next command-line. If the current command-line is the last command line in the history buffer, the next line fetched will be the first command-line in the history buffer.

Before	After
% print foo bar	
% echo Apple II	
% echo GNO/ME	
% _ (DOWN-ARROW)	% print foo bar_
% print foo bar_ (DOWN-ARROW)	% echo Apple II_

Command, Filename and Variable completion

These command-line editor keys can be used to complete filenames, commands and variables.

CTRL-D **list-choices**

Lists commands and pathnames that match the current word.

Before	(CTRL-D)
% print foo_ bar	
After	
foo.c fool foo.m	
% print foo_ bar	

TAB **complete-word**

Command, pathname and variable completion. If the cursor is positioned on the first word of the command-line, command pathname is performed, else pathname or variable completion is performed. The word is expanded to the closest matching command, pathname or variable. Characters are appended up to the point that they would cause more than one. If a complete pathname results for pathname completion, **gsh** appends a "/" if the pathname is a directory; otherwise, it appends a space.

Before	After
% ca_ (TAB)	% cat_
% mo_ foo.c (TAB)	% more_ foo.c
% more fo_ (TAB)	% more foo_
% echo \$TERMC_ (TAB)	% echo \$TERMCAP_

Note that if there is more than one match for the partial command, **gsh** will sound a beep on the speaker. You can use the CTRL-D (**list-choices**) command to see the list of possible matches, and should either finish entering the command manually or type enough additional characters to guarantee a unique match.

If the \$FIGNORE variable is set, **gsh** ignores filenames (when doing completion) that end with any of the suffixes in \$FIGNORE. See Chapter 5 **Description of Pre-defined Shell Variables** for more information regarding the \$FIGNORE variable.

Other ways of entering commands

Terminal Input

An example involving the connection of a terminal; will be dealt with in Chapter 3 **Redirecting Input and Output** but it is necessary to mention here that when using **gsh** over a terminal, some keystrokes must be slightly modified. This is because there are no terminals that can transmit the OA key. Instead, a two-key sequence must be used which replaces OA with ESC. For example, instead of pressing OA-E to toggle insert mode, you can type ESC-E over a terminal to do the same thing.

If you will be using terminals seriously then you should install the Remote Access package.

Script File

While you would normally type commands on the command-line, you can also store a series of often used commands in a file. A file containing such a series of commands is called a script. A script is normally created by using a text editor.

By typing the name of the script file, the shell will execute it, line by line, as if you had typed each command separately. The *gshrc* file presented in Chapter 1 **Customizing the Shell Environment** is an example of a script file.

Chapter 3

Using the GNO Shell more productively

"And then one day, hooray! Another way for gnomes to say hooray!"
Syd Barret, The Gnome

What does this command do?

If you are unfamiliar with what a particular command actually does or what arguments it accepts, you can check quickly by using the electronic manual. GNO/ME includes a utility called **man** which displays the manual pages for a command whose name you supply as an argument. The **man** utility uses another utility called **more** to actually display the pages nicely on the screen. While both of these utilities have electronic manual entries, we have included the printed manual pages in the GNO/ME package to get you started.

Option Arguments

As mentioned in Chapter 2 **Executing Commands**, arguments are passed to a command to extend its usefulness. The arguments presented in the last chapter were words, such as foo, bar and foo.c. Standards exist under unix for programs to accept command-line option arguments. Option arguments (as the name suggests) are optional. There are two standards, short options and long options. Short options are characters that represent commands, whereas long options contain the entire option name.

Consider the following output of the CATALOG command from ProDOS:

/DEV NAME	TYPE	BLOCKS	MODIFIED	CREATED	ENDFILE	SUBTYPE
FINDER.DATA	\$C9	1	21-OCT-91 22:38	14-APR-90 18:24	260	
FINDER.ROOT	\$C9	1	22-OCT-91 17:12	6-OCT-91 15:40	82	
GENESYS	DIR	1	21-OCT-91 23:37	25-APR-91 15:46	512	
GSBUG	DIR	1	21-OCT-91 23:38	19-JUL-90 16:48	512	
MERLIN	DIR	2	22-OCT-91 2:50	30-APR-91 20:21	1024	
LIFEGUARD	\$B3	73	4-SEP-87 4:51	25-DEC-89 20:22	36608	
ORCA	DIR	2	22-OCT-91 17:12	14-SEP-89 18:27	1024	
GNO	DIR	2	22-OCT-91 17:12	13-AUG-91 16:36	1024	
FAST.ANIM	DIR	2	21-OCT-91 23:44	11-MAY-91 10:50	1024	
MICOL	DIR	2	22-OCT-91 3:10	14-JAN-90 2:46	1024	
SRC	DIR	1	21-OCT-91 23:44	7-AUG-91 20:30	512	
NIFTYLIST.V3.3	DIR	2	21-OCT-91 23:44	29-JUL-91 4:04	1024	
MCSRC	DIR	1	21-OCT-91 23:45	7-AUG-91 20:34	512	

BLOCKS FREE:43923 BLOCKS USED:21185 TOTAL BLOCKS:65108

It is impossible to get any variation in the format of this output. While the GNO/ME utility **ls** serves the same purpose as the command CATALOG from Applesoft BASIC, it has a wide number of options which can tailor the output to specific needs. Here is how **ls** can be used to give similar output to the CATALOG command:

```
[1] 2:13am root % ls -l
:dev
total 45k
drw--rd 0000 dir          512 Oct 21 23:45 1991 MCSrc
drw--rd 0000 dir       1024 Oct 21 23:44 1991 NiftyList.v3.3
```



```

drw--rd 0000 dir          1024 Oct 21 23:44 1991 fast.anim
drw--rd 0000 dir           512 Oct 21 23:37 1991 genesys
drw--rd 0000 dir          1024 Oct 22 17:29 1991 gno
drw--rd 0000 dir           512 Oct 21 23:38 1991 gsbug
drw--rd 0000 dir          1024 Oct 22 02:50 1991 merlin
drw--rd 0000 dir          1024 Oct 22 03:10 1991 micol
drw--rd 0100 dir          1024 Oct 22 17:28 1991 orca
drw--rd 0000 dir           512 Oct 21 23:44 1991 src

```

The **-l** short option argument tells **ls** to format the output in long format. **ls** supports only short options. If **ls** did support long options, the above command could be changed to **ls +format=long**. This is clearly more descriptive of what function **ls** will perform. For users new to the UNIX environment, long format options are more user-friendly. However, advanced UNIX users prefer short options because of their brevity.

As indicated above, **ls** has a wide number of options available to format the output. Use the command "**ls -?**" to get a short list of these options. It is left as an exercise for the user to discover how these options affect the output of **ls**. For a complete description of the **ls** command and its options use the command **man ls**.

As an example of the usage and importance of long options, the following is the result of the **+help** option given to the **coff** utility. Note the use of both short and long options:

```
coff [-OPTIONS] filename [segment...] [loadsegment...]
```

OPTIONS	DESCRIPTION
-v [+version]	display coff's version number
-D [+default]	disable default options
-d [+asm]	dump segment body in 65816-format disassembly
-T [+tool]	interpret Toolbox, GS/OS, ProDOS, ROM calls
-x [+hex]	dump segment body in hex (can be used with '+asm')
-l [+label]	print expressions using labels (default is offsets)
-t [+infix]	display expressions in infix form
-p [+postfix]	display expressions in postfix form (default)
-m [+merlin]	format of '+asm' to use merlin opcodes (default)
-o [+orca]	format of '+asm' to use orca/m opcodes
-a [+shorta]	assume 8-bit accumulator for disassembly
-i [+shorti]	assume 8-bit index registers for disassembly
-s [+header]	dump segment headers only
-n [+noheader]	do not print segment headers
-f [+nooffset]	do not print offset into file
-h [+help]	print this information, then quit
filename	name of file to dump
[segment]	names of segments in file to dump
[loadsegment]	names of load segments in file to dump

The long options are much more descriptive, and provide a very easy way to remember options of programs. If an option passed to a shell utility program is not understood by that program, you will generally receive an error message stating that the option is not understood. If the program is user-friendly, a brief list of supported options will also be displayed.

Entering Multiple commands

It is possible to give multiple commands to the GNO shell for processing. To execute multiple commands, place a semi-colon, ";", between them. The commands will be executed sequentially in the order they are entered on the command-line. Take care not to exceed the 1024 character command-line buffer. It is possible to execute multiple commands at the same time, this feature is discussed in Chapter 3 **Background Execution of Commands**.

To run the **echo** command and the **ls** command in succession, enter the following on the command line:

```
% echo Running ls ; ls -l
```

The output of the preceeding command will display the string "Running ls" followed by the output of the "ls -l" command. This method of running several commands in succession is used often to save typing.

Using Aliases as Shorthand

gsh provides a built-in command, **alias**, which allows any command you would type on the command-line to be renamed. You are not limited to renaming a single command name. Rather, you could rename an entire command-line, which could allow you to use the name "backup" to execute the command "backup +source /system +destination /tape.drive". The **alias** command is also a very powerful means of customizing your GNO environment to emulate other computing environments.

To emulate the ORCA environment, the following aliases could be entered into your **gshrc** file, or a script called *orca.alias* that **gshrc** would run:

```
alias copy cp
alias cat "ls -l"
alias catalog "ls -l"
alias move mv
alias rename mv
alias delete rm
alias type cat
alias prefix cd
alias create mkdir
```

If you alias a string containing multiple words, you must enclose the string in quotes, as done for the catalog alias. **gsh** interprets the string as one value. If you do not include both the opening and closing quotes, the alias command will notify you of your error.

You can view any alias' that are set by entering the **alias** command without any arguments. The setting of a particular alias can be viewed by entering one argument consisting of the name of the alias to view.

If you wish to remove an alias, use the command **unalias** with the aliased name as the argument. To remove the aliases from the *orca.alias* file given above, you could do the following:

```
% unalias copy cat catalog move rename delete type prefix create
```

Unlike the **alias** command, the **unalias** command can take multiple arguments. See Chapter 5 **Built-in Commands** for further discussion of the **alias** and **unalias** commands.

Redirecting Input and Output

Most shell utilities write their output to the screen. However, under GNO/ME, like ORCA, it is possible to redirect that output to a file or a GS/OS device. The output of the **ls** command above was imported into this manual by redirecting it to a file. In addition to redirecting the output of a shell utility, it is also possible to redirect the input of that utility. Consider the following **gsh** session:

```
[1]% echo this is a test
this is a test
[2]% echo this is a test > file1
[3]% cat file1
this is a test
[4]% cat < file1
this is a test
```

In the example above, **cat** takes input from *standard input*. In command 3 above, **cat** takes as an argument the filename `file1` and writes the contents of that file to *standard output*. Where no filename argument is given, **cat** reads input from standard input and writes the output to standard output.

In the case of command 4 above, **cat** contains no arguments and therefore reads from standard input. However, **gsh** interprets the "<" redirection operator and opens the file `file1` for use as standard input. Therefore, **cat** will take its input from `file1`, even though it thinks it is reading input from standard input. This input redirection is transparent to the utility, making it work with most shell utilities.

Command 2 above created a new file called `file1`. If this file had existed prior to the command then it would have been erased. It is possible to append output to the end of the file by using the ">>" redirection operator. Consider the following **gsh** session:

```
[5]% echo second line >> file1
[6]% cat file1
this is a test
second line
```

Output that is sent to *standard error*, can also be redirected. The ">&" operator redirects standard error to a file and ">>&" appends standard error to the end of the file. Below is a summary of the redirection operators:

Standard Input	Standard Output	Standard Error	
<			Redirect Input from file
	>	>&	Redirect Output to file
	>>	>>&	Redirect Output to EOF

Output can be redirected to a storage device, printer, modem, or any other valid GNO or GS/OS device. This provides a very powerful means of communicating directly with these devices from within **gsh**. One quick and dirty example of redirection allows a background version of **gsh** to be run on a terminal connected directly through the modem serial port:

```
[1]% gsh < ttya > ttya >& ttya &
```

Pipelines

In addition to the redirection operators, there is one additional operator which gives control over how input and output are handled. The operator is a **pipeline**, "|". Pipelines allow the standard output of one command to be used as the standard input to another command. This is almost equivalent to running the

first command with its output redirected to a temporary file, then running the second command with its input redirected from the temporary file, then removing the temporary file. Pipelines make useful *filter* processes where the output of one command can be sent to another command which filters the output to whatever parameters you give the second command. As an example, you could display all the filenames with the character "a" in their name:

```
[1]% echo foo > file1; echo abc >> file1; echo aabc >> file1
[2]% echo GNO >> file1; echo standard >> file1; echo oof >> file1
[3]% cat file1
foo
abc
aabc
GNO
standard
oof
[4]% cat file1 | grep 'a'
abc
aabc
standard
```

Pipelines are useful when you wish to view lines of text in a file that contain a phrase, or if you want to connect two programs directly, bypassing intermediate files. It is also possible to connect multiple commands with multiple pipelines.

Pipelines are frequently used for paging output. The **coff** program mentioned above prints the output of an OMF disassembly to the screen but does not pause when a key is pressed. In order to pause the display, the output must be piped through a paging utility. The ORCA shell requires that you wait for the entire command to complete execution before the pipeline is processed. However, GNO/ME executes both commands concurrently which allows the **coff** utility to execute while the paging utility displays the program output. GNO/ME comes with two page utilities, **more** and **less**. Complete descriptions of **coff**, **more**, **less** can be found in the electronic manual using the **man** command.

Background Execution of Commands

A major benefit of GNO/ME is *multitasking*. Multitasking is a means of running multiple applications at once (not literally but very close). On the Apple IIGS, GNO/ME accomplishes pre-emptive multitasking by switching among applications that are running in the background. Any GNO/ME utility can be run in the background. Applications running in the background generally run for the same period of time (GNO/ME switches between applications 20 times a second).

To background a shell utility, place the "&" character at the end of the command-line. The GNO shell displays a unique process ID and job number for each backgrounded command.

It is possible to use the background character "&" to separate commands as with the ";" character. Each command with a following "&" is executed in the background.

Up to 32 processes can be executed concurrently under the GNO Kernel.

Warning: When you exit the GNO Shell all processes will be terminated including any you may have running in the background.

Below is a sample session with background tasks:


```

[5] script> ps
  ID  STATE  TT MMID  UID    TIME COMMAND
    1  ready   co 1002 0000   0:45 NullProcess
    2  ready   co 1007 0000   0:05 gsh
 138  running co 1006 0000   0:00 ps
[6] script> cml +p script.c keep=script > outfile &
[1] + 141 Running cml +p script.c keep=script &
[7] script> ps
  ID  STATE  TT MMID  UID    TIME COMMAND
    1  ready   co 1002 0000   0:45 NullProcess
    2  ready   co 1007 0000   0:05 gsh
 141  waiting co 1006 0000   0:00 cml +p script.c keep=script
 142  ready   co 100B 0000   0:00 5/cc
 143  running co 100D 0000   0:00 ps
[8] script> cml +p script.asm keep=script1 > output2 & ps ; ls -s
[2] - 145 Running cml +p script.asm keep=script1 &
  ID  STATE  TT MMID  UID    TIME COMMAND
    1  ready   co 1002 0000   0:45 NullProcess
    2  ready   co 1007 0000   0:05 gsh
 141  waiting co 1006 0000   0:00 cml +p script.c keep=script
 144  ready   co 100E 0000   0:07 5/linker
 145  ready   co 100D 0000   0:00 cml +p script.asm keep=script1
 146  running co 100F 0000   0:00 ps
 147  ready   co 1011 0000   0:00 5/asm65816
    3  barf           1  outputfile    6  script.asm    1  script.root
    1  foobar          19  script        3  script.c     36  script.sym
    1  output2         6  script.a      6  script.mac    1  typescript
[9] script> cp script.asm script2 &
[3] 150 Running cp script.asm script2 &

[2] - Done      cml +p script.asm keep=script1 &
[1] + Done      cml +p script.c keep=script &
[3] - Done      cp script.asm script2 &

[10] script> ps
  ID  STATE  TT MMID  UID    TIME COMMAND
    1  ready   co 1002 0000   0:45 NullProcess
    2  ready   co 1007 0000   0:05 gsh
 151  running co 1006 0000   0:00 ps

```

The first command line sends the `ps` command to the shell. `ps` lists the processes currently being executed by the GNO kernel. The processes named `gsh` and `NullProcess` are always present. For a complete description of the `ps` command see Chapter 4 **Kernel Commands**.

When a command is executing in the background, keyboard input is not sent to it. However, output is still treated in the same way. If the command sends output to the standard output or standard error, the screen will become cluttered. Try this example:

```

[1]% ls -l&
[2]% ls -l

```

Both the output of commands #1 and #2 will be sent to the screen. After command #1 is entered and you begin typing command #2, you will see the output of the first "`ls -l`" command being sent to the screen while you enter command #2. Utilities which produce output should have their standard output and standard error redirected to a file when they are executed in the background. See Chapter 3 **Redirecting Input and Output**.

Executing commands in the background hinders the performance of the Apple IIGS. This is not too noticeable when one or two commands are being executed but performance will degrade more noticeably as more commands are started. The Apple IIGS was not designed as a multitasking computer so the performance of GNO/ME should be understandable. If you have an accelerator (such as the Transwarp GS or Zip GS) installed, performance of multiple tasks will be acceptable.

Job Control

Now that command backgrounding and multitasking have been discussed, some more definitions can be mentioned. A process is a command which has been submitted to the shell for execution. **Gsh** contains a set of special commands which make dealing with processes much easier. **gsh** treats each command entered at the command-line as a **job**, where a single job may contain multiple processes. For example:

% ls	{one command, one process, one job}
% ls ; ps	{two commands, two processes, two jobs}
% ls & ps	{two commands, two processes, two jobs}
% ls more	{two processes, one job}

When a job is run from the shell, it can be in several modes of operation. Jobs can be in any of three states: **running**, **stopped**, or **done**. A job can be executing in either the foreground or the background.

Commands exist to place a job in any mode of operation. When a job is run directly from a command-line it is running and it is in the foreground. Since the command-line cannot be accessed, two special keys have been defined: **^c** kills the job and **^z** will stop the job. When the job is killed, it is gone forever, but a stopped job can be restarted. When a job is stopped, the kernel suspends each of the processes in the job.

Jobs that are running in the background or have been stopped can be accessed using several built-in commands. The **bg** command will place a job in the background, placing it in the running state if necessary. The **fg** command will similarly place a job in the foreground, and the **stop** command will stop a backgrounded job. The **kill** command will terminate a job.

Each time **job control** is accessed, a special job status line is displayed following the command. The first item on the left in brackets is the job number. Next is a single character, either a '+', '-', or a blank. The '+' designates the currently accessed job, the '-' is the previously accessed job, and all other jobs are not specified. The **jobs** command will display a list of all jobs.

This example was used in Chapter 3 **Background Execution of Commands**, but now more of the notation will be understandable

```
[5] script> ps
  ID  STATE  TT MMID  UID    TIME COMMAND
    1  ready   co 1002 0000   0:45 NullProcess
    2  ready   co 1007 0000   0:05 gsh
 138  running co 1006 0000   0:00 ps
[6] script> cml +p script.c keep=script > outputfile &
[1] + 141 Running cml +p script.c keep=script &
[7] script> ps
  ID  STATE  TT MMID  UID    TIME COMMAND
    1  ready   co 1002 0000   0:45 NullProcess
    2  ready   co 1007 0000   0:05 gsh
 141  waiting co 1006 0000   0:00 cml +p script.c keep=script
 142  ready   co 100B 0000   0:00 5/cc
 143  running co 100D 0000   0:00 ps
[8] script> cml +p script.asm keep=script1 > output2 & ps ; ls -s
[2] - 145 Running cml +p script.asm keep=script1 &
```



```

ID  STATE  TT MMID  UID    TIME COMMAND
1   ready   co 1002 0000   0:45 NullProcess
2   ready   co 1007 0000   0:05 gsh
141 waiting co 1006 0000   0:00 cml +p script.c keep=script
144 ready    co 100E 0000   0:07 5/linker
145 ready    co 100D 0000   0:00 cml +p script.asm keep=script1
146 running co 100F 0000   0:00 ps
147 ready    co 1011 0000   0:00 5/asm65816
3   barf      1 outputfile 6 script.asm 1 script.root
1   foobar     19 script      3 script.c   36 script.sym
1   output2    6 script.a    6 script.mac 1 typescript
[9] script> cp script.asm script2 &
[3]      150 Running cp script.asm script2 &

[2] - Done    cml +p script.asm keep=script1 &

[1] + Done    cml +p script.c keep=script &

[3] - Done    cp script.asm script2 &

[10] script> ps
ID  STATE  TT MMID  UID    TIME COMMAND
1   ready   co 1002 0000   0:45 NullProcess
2   ready   co 1007 0000   0:05 gsh
151 running co 1006 0000   0:00 ps

```

Each of the special commands, bg, fg, stop and kill, take an argument which specifies the job to perform the operation on. The argument is either a number specifying the process id, or a '%' followed by one of the following: '+' or '-' for the current job, a '-' for the previous job, or a number to specify any specific job. If nothing follows the '%' or the argument is missing, then the current job is the default.

```

[1] gno> cat gshrc

###
#
# GNO 2.0 gshrc file
#
###
#
# Initialize our environment
#
set path=":right:gno:bin :right:gno:usr:bin"
set prompt="[h] %S%t%s %C> "
set home=":right:gno:user:root"
set term=gnocon
export path prompt home term
setenv history=100 savehist=25
###
#
#Set up standard prefixes for utilities.
#
###
prefix 2 :software:orca:libraries
prefix 3 :software:orca
prefix 4 :software:orca:shell
prefix 5 :software:or^Z
[1] + Stopped (signal) cat gshrc
[2] gno> jobs

```



```

[1] + Running cat gshrc
[3] gno> bg           {output was not redirected, so screen gets cluttered}
[1] + Running cat gshrc
ca:languages
prefix 6 :software:orca:utilities
prefix 7 :tmp
###
#
# Set up prefixes for Orca2.0(tm)'s benefit
#
###
prefix 13 :software:orca:libraries
prefix 14 :software:orca
prefix 15 :software:or
[4] gno> stop           (Incredibly fast typing :- )
[1] + Stopped (signal) cat gshrc
[5] gno> jobs
[1] + Running cat gshrc
[6] gno> fg
[1] + Running cat gshrc
ca:shell
prefix 16 :software:orca:languages
prefix 17 :software:orca:utilities
alias ls 'ls -CF'
alias dir 'ls -al'
alias cp 'cp -i'
alias rm 'cp -p rm'
alias mv 'cp -p mv'
setenv usr^Z
[1] + Stopped (signal) cat gshrc
[7] gno> jobs
[1] + Running cat gshrc
[8] gno> kill %1
[9] gno> jobs
[10] gno>

```

There is one additional way that a job may be stopped. If the job is placed in the background and it attempts to read from the console, the job will be stopped, and the status line says "(tty input)" as the reason for the job being stopped. The job should be foregrounded so that the user may enter input to the program. It can then be placed back in the background as necessary (with ^Z and **bg**).

Working with Pathnames

To move easily to directories descended from the home directory, **gsh** provides the "~" (tilde) character. This character represents the home directory. Therefore, if your home directory was ":hard:gno:user:root", you could use the command "**cd ~**" to move to the home directory (note that **cd** without any arguments also defaults to the home directory). To move to subdirectories of the home directory, you could use the command "**cd ~/dir1**" command. The tilde character is recognized by **gsh** before the command is interpreted.

Another special sequence "...", when used as part of a pathname, will strip the last path between pathname separators. For example, the pathname "/dev/gno/.." would be expanded to "/dev". The "/gno" path is stripped as it is before the periods. This provides an excellent way to backup into your directories. "Backing up" is limited by the volume directory of the device being used.

Additionally, the character ".", can be used to signify the current directory.

Examples:

```

[/dev/gno]% cd ~                               { change to home directory }
[/dev/gno]% cd ~/src
[/dev/gno/src]% cp ~/file1 file2                { copy /dev/gno/file1 to /dev/gno/src/file2 }
[/dev/gno/src]% echo ../
/dev/gno/..
[/dev/gno/src]% cd ../
[/dev]% mkdir ~/utilities                       { create directory /dev/gno/utilities }
[/dev]% mkdir ./libraries                       { create directory /dev/libraries }
[/dev]% cd ~
[/dev/gno]% cp src/file1 .                      { copy /dev/gno/src/file1 to /dev/gno/file1 }

```

Pathname Expansion

Many utilities supplied with **gsh** take, as an argument, a filename or filenames. The shell utilities **cat**, **ls**, **grep**, and **cp** can take multiple filenames as arguments. If you wish to invoke any of these utilities on filenames that have a sequence of characters in common (ie. AND, APPLE, SHK, TXT, FILE2, FILE3, etc.), **gsh** provides special characters, called regular expressions or wildcards, which match multiple filenames without having to enter all filename arguments manually.

*	Matches any string of characters.
?	Matches a single character.
[abc]	Matches any of the characters enclosed in brackets.
[^abc]	Matches any of the characters not enclosed in brackets.
[a-c]	Matches the ascending sequence of characters enclosed in brackets.

This method of matching filenames is known as **globbing**. **gsh** performs globbing on the word prior to executing the command. The following **gsh** session illustrates file globbing:

```

[1]% cd /dev/gno/utilities
[2]% ls
:dev:gno:utilities
CONV          Crunch          CrunchIIGS      DeRez          DiskCheck
DumpObj       Duplicate         EMACS          Equal          Express
Files         LinkIIGS          MakeBin        MakeDirect     OrcaDumpIIGS
Prizm         ResEqual         Search         canon          choose
clrff        cmdfix           coff           compact        count
detab         dir              dirff          dumpfile       eject
emacs.doc     emacs.hlp        emacs.rc       emacs.tut      help
init          join             link           macgen         makelib
mem           online           pageject      pause          pwd
src
[3]% ls e*
:dev:gno:utilities
EMACS          Equal          Express         eject          emacs.doc
emacs.hlp      emacs.rc       emacs.tut
[4]% echo *r *m
dir Prizm mem
[5]% echo *i*
cmdfix CrunchIIGS Prizm DiskCheck Duplicate Files init join LinkIIGS makelib
MakeBin MakeDirect link dirff dumpfile online OrcaDumpIIGS dir
[6]% echo NoMatch*
No match.
[7]% echo [a-f]*

```



```

coff canon cmdfix compact Crunch CrunchIIGS DeRez DiskCheck DumpObj Duplicate
EMACS emacs.doc emacs.hlp emacs.rc emacs.tut Equal Express Files choose clrff
count detab CONV dirff dumpfile eject dir
[8]% echo [a-fs-t]*
coff canon cmdfix compact Crunch CrunchIIGS DeRez DiskCheck DumpObj Duplicate
EMACS emacs.doc emacs.hlp emacs.rc emacs.tut Equal Express Files choose clrff
count detab Search src CONV dirff dumpfile eject dir
[9]% echo emacs?*
EMACS emacs.doc emacs.hlp emacs.rc emacs.tut
[10]% echo [^a-f]*
Prizm help init join LinkIIGS makelib MakeBin MakeDirect link mem ResEqual
Search src online pageeject pause OrcaDumpIIGS pwd macgen
[11]% echo [^a-fs-t]*
Prizm help init join LinkIIGS makelib MakeBin MakeDirect link mem ResEqual
online pageeject pause OrcaDumpIIGS pwd macgen
[12]% echo ???
mem src pwd dir
[13]% echo ?
No match.
[14]% echo "???"
???
[15]% do you have a light?
No match.

```

As can be seen by the above example, character matches are case insensitive. The ProDOS file system treats the filenames "file" and "FILE" as the same file. **gsh** recognizes this and does not detract from the underlying file system.

File globbing makes passing arguments to commands much easier and much more powerful. You could easily use `*.c` as an argument in a number of ways:

```

[1]% ls *.c                                { lists all filenames ending in ".C" }
...
[2]% cc *.c                                { compiles all files ending in ".C" }
...
[3]% more *.c                              { displays contents of all files ending in ".C" }
...

```

Quoting Special Characters

Beginning with Apple IIGS System Software 6.0, GS/OS is able to read files from Macintosh computers. The Macintosh uses a filesystem known as HFS, which allows filenames to contain any character except the colon (":"). Because a filename such as "emacs?*" is valid under HFS, care must be taken or unexpected results will occur. The word "emacs?*" was used as a regular expression above to specify a list of filenames beginning with the word "emacs" and one or more trailing characters. **gsh** does provide a way to pass an argument which contains special shell characters to a command. This is known as quoting an argument. There are three different ways to quote an expression:

The single quote will quote everything between the single quote marks. Thus, to display the contents of a file on an HFS volume named "emacs?*":

```
% more 'emacs?*'

```

The double quote will quote everything between the double quote marks except variables. See Chapter 5 for more on variables.


```
% echo "emacs?* $home"
emacs?* /dev/gno
```

The backslash is used to quote one character. To pass "emacs?*" as a regular expression using the backslash, enter the following:

```
% ls emacs\?\*
```

One additional purpose of the quoting mechanism built into **gsh** is to add spaces to command arguments. Each command and its arguments is separated by a space. Multiple spaces between arguments are treated as one space. Thus, consider the following:

```
% echo a      b c
a b c
% echo 'a      b c'
a      b c
```

How gsh Finds a Command

gsh has a special variable, **PATH**, which specifies the directories and order of directories to search for shell utilities. This variable is often setup in the *gshrc* file although it can be changed as often as needed. The purpose of the **PATH** variable was discussed in Chapter 1 **Customizing the Shell Environment**.

When **gsh** starts up, it searches all directories specified in the **PATH** variable and establishes a table of all commands, called a hash table. Because of this table, **gsh** "knows" where a command is and can execute the command much faster than searching through all directories every time the command is entered.

The search process begins with alias names. See Chapter 3 **Using aliases as shorthand**. If an alias is found that matches the command, the alias is replaced with its value and the command-line is again parsed. If it was not an alias, **gsh** checks to see if it was a special built-in utility. The search process then searches for the name in the hash table. If an entry is found in the hash table, the path name of the command is retrieved and the command is executed. If an entry is not found, the current path is searched. If the command name is not found, an error results.

When the **PATH** variable is changed, **gsh** does not automatically recreate the command hash table. You need to issue the command **rehash** to recreate the hash table. The more pathnames specified, the greater the delay in starting **gsh** and in invoking the **rehash** command. The following shell script changes **PATH** and invokes the **rehash** command in one step.

```
echo Resetting PATH variable $PATH to $1
set path=$1
rehash
```

The **\$1** variable will be expanded with the first argument passed to the script.

rehash should also be used if a new utility is copied to one of the directories specified in the **PATH** variable. Of course, it is possible to specify the absolute pathname of any command, but this is undesirable if the command is frequently used.

Chapter 4

Built-in Commands

Built-ins vs EXE Commands

The term "built-ins" is used to describe commands that exist within the shell itself. These utilities run faster than external commands because the code is already loaded into memory. Files of type "EXE", on the other hand, must be loaded into memory by **gsh** and executed. If an EXE command is executed again, it might, again, have to be loaded into memory. This results in longer execution time for the program.

gsh has a number of built-in commands which allow you to work with the shell, the GNO kernel, and the shell environment.

The following section describes the commands that are built-in to **gsh**. The "[..]" character sequence represents an optional argument to a command. The term "SIGNAL" is used to represent one of the signal names or numbers listed in Appendix D **Signals**. The sequence "..." means the command accepts multiple arguments of the same type as the argument before the "..." sequence. The sequence "{..}" is used to represent a set, which is a list of possible arguments to choose from.

Shell Commands

Shell built-ins provide support for external shell commands (i.e. EXE files and shell scripts) and provide some commands used in every-day work.

bindkey [-l] function string

Bindkey is used to customize the shell's command-line editor. Any key on the keyboard can be mapped to any of a number of functions. The various functions are as follows:

backward-char	move cursor left
backward-delete-char	delete character to left
backward-word	move cursor left one word
beginning-of-line	move cursor to beginning of line
clear-screen	clear screen and redraw prompt
complete-word	perform filename completion
delete-char	delete character under cursor
down-history	replace command line with next history
end-of-line	move cursor to end of line
forward-char	move cursor to the right
forward-word	move cursor one word to the right
kill-end-of-line	delete line from cursor to end of line
kill-whole-line	delete the entire command line
list-choices	list file completion matches
newline	finished editing, accept command line
raw-char	character as-is
redisplay	redisplay the command line
toggle-cursor	toggle between insert and overwrite cursor
undefined-char	this key does nothing
up-history	replace command line with previous history

Keys are bound to functions, not vice-versa. This means that you can have any number of commands refer to the same function. For example, the default bindings have CTRL-A and OA-< both bound to beginning-of-line.

Most of the function names are self-explanatory, and are explained in Chapter 2, but a few deserve discussion. `raw-char` is what you should bind a key that should be inserted into the command-line as-is. The regular printable ASCII set, such as the letters a-z, numbers, etc. are bound to `raw-char`. Control characters should not be bound to `raw-char` because the command-line editor will become confused (most control characters act as special GNO/ME console feature codes - see the *GNO Kernel Reference Manual*, Chapter 4 **TextTools Replacement**).

Any keystroke that should be rejected by the editor should be bound to `undefined-char`. By default, this includes control characters and OA-sequences that are not assigned to any editing features. Any key bound to `undefined-char` will cause `gsh` to beep and ignore the key.

You can actually bind key sequences, not just keystrokes, to functions. There is no limit other than memory to how many characters are in a command sequence.

Because terminals do not have the OA (Open Apple) key, OA is actually mapped by the kernel to a two-character sequence consisting of ESC and the key. For example, OA-Y would actually produce ESC-Y.

Control characters in the `string` are represented in ^X format; e.g. CTRL-A is represented by ^A. ESC (and OA) is represented by ^[.

Examples:

```
% bindkey kill-end-of-line ^K  map CTRL-K to kill-end-of-line (like Emacs)
% bindkey clear-screen ^[^X    map OA-CLEAR to clear-screen
```

commands

Displays a list of all built-in shell commands.

cd [pathname]

chdir [pathname]

Changes the current working directory to `pathname`. If `pathname` is not given, the default HOME directory (i.e. the value of the HOME environment variable) is used. This makes it easy to move back to your home directory. Under `gsh`, unlike most UNIX shells, the `cd` is not necessary, except to change automatically to your HOME directory. If a command is not a built-in or EXE file, but is instead the name of a directory, a `cd` is implied and performed on the directory unless the `NODIREXEC` variable has been set.

```
[1] gno> cd utilities
[2] gno/utilities> echo $HOME
/dev/gno
[3] gno/utilities> cd
[4] gno> utilities
[5] gno/utilities> ../utilities
[6] gno/utilities> ~
[7] gno> _
```

clear

This command takes no arguments. When invoked, the screen will be cleared.

df

This command takes no arguments. When invoked, a listing of free blocks for every block device is given. In addition, the device name, type, file system, and capacity is listed.


```
[1] gno> df
```

.d##	Volume	Device	Free	Total	Capacity	System
.d1	:Procyon	.CVTECH.S6.A	3166	41037	92%	ProDOS
.d2	Apple 3.5 Drive	.APPLEDISK3.5A				
.d3	Apple 3.5 Drive	.APPLEDISK3.5B				
.d4	Console Driver	.CONSOLE				
.d5	:Day	.CVTECH.S6.B	15881	65535	75%	ProDOS
.d6	:Night	.CVTECH.S6.C	39274	65535	40%	HFS
.d7	:Left	.CVTECH.S6.D	5365	51776	89%	ProDOS
.d8	:Right	.CVTECH.S6.E	27477	65535	58%	ProDOS
.d9	:Software	.CVTECH.S6.F	3289	40960	91%	ProDOS
.d10	:RAM5	.RAMDISK	505	512	1%	ProDOS
.d11	:Conner 40	.AFP1	7368	80604	90%	AppleShare
.d12	AppleTalk fsd	.AFP2				
....						
.d24	AppleTalk fsd	.AFP14				
.d25	AppleTalk RPM	.RPM				
.d26	AppleTalk Main	.APPLETALK				
.d27	Serial Modem	.SERIAL2				
.d28	:system	.APPLESCSI.HD01.	27561	50773	45%	ProDOS
.d29	:dev	.APPLESCSI.HD01.	43431	65108	33%	ProDOS
.d30	:usr	.APPLESCSI.HD01.	44376	48725	8%	ProDOS

Volume - Name of device.

Device - GS/OS device name.

Free - Total number of free blocks on device.

Total - Total number of blocks on device.

Capacity - Percentage of used blocks on device.

System - Format of file system. With System Software 6.0.1, the file systems available are ProDOS, HFS, Pascal, MS-DOS, and DOS 3.3, and some CD-ROM formats.

```
echo [-n] arg [arg ...]
```

Expands the "arg" expression(s) and outputs them to the screen. If the "-n" switch is specified, a NEWLINE character is not output after the last "arg" expression. Special escape sequences may also be included in the arguments, similar to those used in C strings:

\b - Backspace

\f - Clears screen (form feed)

\n - Newline

\r - Return

\t - Tab

\nnn - A decimal ASCII code. nnn represents the ASCII code.

```
[1] gno> echo Hello World
```

```
Hello World
```

```
[2] gno> echo -n Hello World
```

```
Hello World[3] gno> echo $PATH $HOME 'come get to $gnome'
```

```
/dev/gno/utilities /dev/orca/utilities /dev/gno come get to $gnome
```

```
exit
```

Exits the shell or terminates a shell script.

```
history
```

This command displays the list of previous command-line entries. The number of entries saved is set in the \$HISTORY variable.

pushd popd [+n] dirs

These three commands maintain the shell's *directory stack*. Let's say you're working in a directory `/src/myprogs/class/program.1/`, and you want to temporarily go to another directory. Instead of having to 'cd' there and 'cd' back to a very long directory name (i.e., lots of typing), you can use the `pushd` command, like so:

```
gsh> pushd /etc                { start in /src/myprogs/class/program.1/ }
...
gsh> popd                      { back in /src/myprogs/class/program.1/ }
```

The `pushd` command stores the current directory on a stack, and then changes the current directory to the argument. When you want to go back to the original directory, type `popd`. The shell will pull the last directory off the stack and 'cd' to that directory.

The `popd` command when given an argument of `+n` will remove the `n`'th directory from the stack. It does not change to that directory.

The `dirs` command displays the current directory stack.

pwd

Displays the current working directory. This is useful if you have not configured the `$PROMPT` string to output your current working directory.

```
[1]> cd $HOME
[2]> pwd
/user/root
[3]> cd utilities
[4]> pwd
/user/root/utilities
```

source

When a script is executed, `gsh` creates a new process to run the script. As a result, scripts cannot change the shell's prefixes and certain other parameters. Instead of executing the script directly, you may use the `source` command which does not create a new process to execute the script. Thus, the `source` command is effectively exactly like typing all the commands in the script from the keyboard.

tset

The `tset` command causes the shell to reread the `/etc/termcap` file and reset its output system to use the terminal type specified in the `$TERM` variable. On startup, after reading the `gshrc` file, `gsh` automatically does a `tset`. `gsh` also automatically does a `tset` whenever the `$TERM` variable is changed with the `set` command. You would use `tset` manually if, for example, a utility changed the value of `$TERM`.

which command_name [command_name ...]

Let's say that you are working on a new version of the venerable shell utility `ls`. Since a search of the hash table is done before searching the current directory, you might accidentally be using the wrong version of the command. You make changes and run the new program, but your changes don't seem to appear! Use the `which` command to check your sanity. Which also comes in handy in locating duplicate program names in the `$PATH` directories (for example, an `ls` in both `/bin` and `/usr/bin`.)

The way to access a utility in the current directory which has the same name as a program in the `$PATH` is to prefix the command name with `./`, as in `./ls`. Also, see `unhash`.

Kernel Commands

gsh provides a set of commands to control the GNO kernel. These commands mainly deal with job control. See Chapter 3 **Job Control**.

bg (%job | pid)

Starts the specified job, if stopped, and places it in the background.

fg (%job | pid)

Starts the specified job, if stopped, and brings it into the foreground.

jobs [-l]

Displays a list of the shell's jobs. If the "-l" switch is specified, the process id is included in the job list.

```
[1] gno> cml foo.c keep=foo &
[1] + Running cml foo.c keep=foo &
[2] gno> echo hello
hello
[3] gno> jobs
[1] + Running cml foo.c keep=foo &
```

kill [[-SIGNAL] %job | pid | [-l]

The kill command will send the signal SIGNAL to the process number pid. The ps command documented below describes how to list all process ID's currently executing.

SIGNAL can be either a numeric value or string representing the signal to be sent to the process. All signals are documented in Appendix D **Signals**, with numeric and string value listed. Alternatively, specifying the -l option will list all the signals and their names.

If the process number isn't known, but the job number is, replace the pid with a '%' followed by the job number.

ps

This command takes no arguments. When invoked, a list of all currently running processes is displayed.

```
[2] 9:52pm root> ls -lR :hard:gno > /ram5/dev &
[1] + 35 Running ls -lR :hard:gno &
[3] 9:53pm root> ps
  ID  STATE  TT MMID  UID    TIME COMMAND
   1  ready   co 1002 0000   0:26 NullProcess
   2  ready   co 1005 0000   0:02 gsh
  35  ready   co 100A 0000   0:01 ls -lR :hard:gno
  36  running co 1007 0000   0:00 ps
[4] 9:53pm root>
[1] + Done ls -lR :hard:gno
```

ID - A unique process ID assigned to a command by GNO. Use this number to reference any process.

STATE - Current state of the process. Each process can be in any of the following states:

RUNNING	the process is currently in execution.
READY	the process is not currently executing, but is ready to be executed as soon as it is assigned a time slice.
BLOCKED	the process is waiting for a slow I/O operation to complete (for instance, a read from a TTY).
NEW	the process has been created, but has not executed yet.

SUSPENDED	the process was stopped with SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU.
WAITING	the process is waiting on a semaphore 'signal' operation. Programs waiting for data from a pipe have this state.
WAITSIGCH	the process is waiting to receive a SIGCHLD signal.
PAUSED	the process is waiting for any signal.

TTY - Terminal connected to process.

MMID - Memory Manager ID assigned to process.

UID - ID of the user who initiated the process.

TIME - How much CPU time this process has used. This is not the elapsed time of the process.

COMMAND - Command-line string used to invoke process.

setdebug (val | {+|-}flag)

Turns GNO kernel debugging code on or off. The value passed consists of a bit field, where each bit specifies a different type of debugging code to activate. An alternate method is to provide a list of debug flags, either preceded by a '+' or a '-'. Those preceded by a '+' are activated, and those preceded with a '-' are deactivated. All debugging is deactivated by passing a value of 0. Running setdebug with no arguments returns a list of the debugging flags. Legal flags include:

- gsostrace - Trace GS/OS calls
- gsosblocks - Trace GS/OS parameter blocks
- gsoserrors - Trace GS/OS errors
- pathtrace - Trace GS/OS pathnames
- sigtrace - Trace signals
- systrace - Trace system calls

stop (%job | pid)

Stop the execution of all processes in a specified job.

Environment Commands

The last set of commands, environment commands, modify the **gsh** environment. Many of these commands have been used in other parts of this manual and, therefore, should not be new.

alias [name] [value]

Creates an alias for a string. When this alias is referenced as a command, **value** will be expanded into the command line. For commands that require many arguments or have several steps, you could set up an alias to save typing. You can also use aliases to create new names for commands. To obtain a list of all aliases, invoke **alias** with no arguments. To list the value of an alias, invoke **alias** with name only. Here are some alias examples:

<u>Alias</u>	<u>Command Name</u>
ll or list_long	ls -l
backup_sys	cp -r /system /BackupDrive&
backup_home	cp -r \$HOME /BackupDrive&
print	echo
catalog	ls -l
delete	rm
copy	cp
type	more
rename	mv

export [variable ...]

When a shell environment variable is marked as exportable, any process that is created from within the current process (most likely **gsh**), will be passed copies of the exported variables. See **setenv** and Chapter 5 **Scope of shell variables**.

hash

Displays a list of all commands currently in the shell's hash table; i.e., a list of commands in the various **\$PATH** directories.

prefix [prefixnum [prefixname]]

GNO maintains a list of 32 'prefixes' for each process. Prefixes allow the user to reference a directory with a number. While **gsh** provides this ability with environment variables, the **prefix** command exists to support the ORCA compilers and other utilities that are dependent on certain GS/OS prefixes. Appendix B contains a list of these prefixes and their "default" meanings, as documented in the *"Apple IIGS GS/OS Reference"*, Volume 1.

If **prefixname** is not given, the value of **prefixnum** is displayed. If neither argument is given, a list of currently assigned prefixes is displayed.

rehash

To decrease the time spent searching for a command, **gsh** builds a table of all commands which were found in the pathnames set in the **\$PATH** variable. When a command is invoked, only this list is searched. When the **\$PATH** variable is changed, **gsh** must rebuild this list. The **rehash** command tells **gsh** to rebuild the list.

While the old list is still active, if **\$PATH** is changed and one of the previous search paths is no longer online, **gsh** will try and execute the command from the offline device, resulting in a command failure.

To make this a one-step process, the **change.path** shell script listed in Chapter 5, **Accessing shell variables** can be used.

```
set [var] [value] [{var value} ...]
set var=value [var=value ...]
setenv [var value]...
```

Use these command to create or modify environment variables. If **set** is invoked with no arguments, a list of the current environment variables is displayed. If only **var** is given as an argument, the value of **var** is displayed. To set or reset a variable, use both the **var** and **value** arguments. There are two ways to set a variable, either by "**var value**" or "**var=value**". To set multiple variables at once, simply list them all on the command line as shown above.

setenv works just like **set**, but automatically exports the variable(s) or lists only exported variables.

When using **set** or **setenv** to view a list of variables, exported variables appear in ALL CAPS.

unalias name [name ...]

To remove an alias from the alias list, use this command. To remove multiple aliases with one command, specify all the aliases on the command line.

unhash

To disable the internal hash table created with the **rehash** command, use this command. This is useful if you wish to use only utilities in the current working directory (during testing, for example).

unset var [var...]

To remove a variable from the environment, use `unset`. `unset` accepts multiple names if more than one variable is to be deleted. Future attempts to access the variable `var` will result in an error or a NULL string, depending on the circumstances.

```
[8] 9:57pm root> set
PAGER = less
PATH = :hard:gno:bin :right:gno:usr:bin
user1 = foo
USRMAN = /usr/man
HISTORY = 30
status = 227
figignore = .a .root .sym
TERM = gnocon
LESS = -e
PROMPT = [%h] %S%t%s %C>
HOME = :hard:gno:user:root
user = user1
nonewline = 1
[9] 9:57pm root> unset user1
[10] 9:57pm root> unset user
[11] 9:57pm root> set
PAGER = less
PATH = :hard:gno:bin :right:gno:usr:bin
USRMAN = /usr/man
HISTORY = 30
status = 0
figignore = .a .root .sym
TERM = gnocon
LESS = -e
PROMPT = [%h] %S%t%s %C>
HOME = :hard:gno:user:root
nonewline = 1
```


Chapter 5

Shell Variables

And then one day, hurray! Another way for gnomes to say hooray!
Syd Barret, The Gnome

Using shell variables

gsh supports **variables** in the shell environment. These variables can be used by any shell utility or script. Many EXE files and shell scripts pre-define certain shell variables that contain formatting options or other options for a specific utility. As an example, the **ls** utility looks for the variable **TERM** that defines the terminal type currently being used. When **ls** is started, it reads the value of the **TERM** variable and avoids printing Apple II specific MouseText characters if the set terminal does not support them.

gsh has set aside certain variables for its specific use. Shell utilities should be aware of these variables and use them appropriately. Use caution when changing shell variables, because the change could affect more than just the shell.

Scope of Shell Variables

There are two types of processes that are involved in any discussion of a multitasking system. The original process, **gsh** for example, is called a parent process. If **gsh** invokes a process, such as **ls**, **cp**, or **mv**, that process is called a child process. It is possible for any process to define a variable. These variables will not be made available to other processes unless the program that defined the variable specifically makes them available.

The **export** command makes variables defined by one process available to its children. See the example **gshrc** shell script shown in Chapter 1 **Customizing the Shell Environment**. In the case of the shell, most of its variables are exported and, therefore, all shell utilities can read the value of a shell variable. However, programs cannot change the value of a shell variable. In general, executables share their environment with that of the shell, so that a utility can change variables in its parent's environment. This allows communication between programs and the shell.

Description of Pre-defined Shell Variables

The following variables have special meaning to **gsh**. Shell variables are not case sensitive.

- \$0, \$1, \$2, ...** String values that contain the arguments to a shell script. Variable 0 contains the name of the script. The first argument begins with variable 1 and so on. To expand a variable, use the dollar character, "\$" (See Chapter 5 **Accessing shell variables**)
- \$<** When encountered, the variable is expanded using a value obtained from standard input. This provides a means of obtaining user input in script files. Note that the shell variables are expanded before the command-line is executed (See Chapter 5 **Accessing shell variables**) When prompting the user for input, be sure that the prompt is in a separate command-line than the **\$<**. Also, if the user wishes to enter a value with spaces, he must quote what he types with double-quotes.

\$ECHO	A boolean value that, if defined, will cause commands in a shell script to be echoed to standard output.
\$FIGNORE	This variable, if set, contains a list of filename suffixes. When doing command or filename completion, gsh will ignore any filename with a suffix listed in FIGNORE. For example, you might want to set <code>fignore=".A .ROOT .SYM"</code> to ignore object files and other compiler droppings.
\$HISTORY	A numeric value that contains the number of history commands (command-lines) remembered. If the value is 0 or HISTORY is undefined, all commands will be remembered. Previous command-lines can be called back with the UP-ARROW and DOWN-ARROW (See Chapter 2, History Editing Commands).
\$HOME	The HOME directory is the main directory of the shell; it is the directory gsh defaults to when it starts. The tilde character ("~") can be used as a shorthand method of accessing the HOME directory (as discussed in Chapter 3, Tilde Expansion).
\$IGNOREEOF	A boolean value that, if enabled, will prevent ^D from exiting the shell.
\$NOBEEP	A boolean value that, if set, will prevent gsh from sounding the speaker when errors occur while editing a command-line.
\$NODIREXEC	A boolean value that, if set, will disable gsh 's feature of treating directory names as commands; i.e. if a directory is specified as a command, gsh will move to that directory as though the <code>cd</code> command was being used.
\$NOGLOB	A boolean value that, if enabled, will disable filename globbing. Command arguments will be passed to their commands "as-is", without any wildcard expansion.
\$NONEWLINE	A boolean value that, if set, will disable carriage returns being output before and after command execution. Previous examples given in this manual have this option set.
\$PATH	A string value that defines the pathnames where shell scripts, EXE utilities, and SYS16 programs can be found (See Chapter 3, How gsh finds a command). Because GS/OS uses colons as separators in pathnames, gsh cannot allow colons to be used as separators in the PATH variable, as UNIX does. If one of the path entries has a space within it (which is possible with the HFS FST), then the space should be quoted with a backslash, "\".
\$PRECMD	Actually an <u>alias</u> and not a variable, if PRECMD is defined it will be executed just before gsh prints the prompt for a command line. For example, <code>alias precmd qtime</code> will print the time in English text before every prompt.
\$PROMPT	When gsh prompts you to enter a command, the prompt that appears before the cursor can be customized for your gsh environment. If PROMPT is undefined, the default prompt of "% " is used. The prompt string recognizes certain character sequences in the PROMPT variable and interprets them accordingly. The following are the special characters: %h, %!, ! Current history number. %t, %@ Current time of day in 12 hour am/pm format. %d, %/ Current working directory. %~ Current working directory with tilde ("~") replacement. %c, %C, % . Trailing component of current working directory. %S, %s Inverse mode ON (%s) / OFF (%s).

<code>%U, %u</code>	Begin and end Underline mode (only on terminals that support underline. <code>gnocron</code> will use inverse mode instead.)
<code>%%</code>	Displays a single "%" character.
<code>%n</code>	User name (contents of <code>\$USER</code> variable).
<code>%W</code>	The date in mm/dd/yy format.
<code>%D</code>	The date in yy-mm-dd format.
<code>\n</code>	Newline
<code>\r</code>	Carriage Return
<code>\t</code>	Tab
<code>\b</code>	Bell (beep)

Examples of prompts already used in this manual are "[1]%", and "[/user/root]%" These were created by using the commands **set prompt "[%h]%"** and **set prompt "[%d]%"**.

`$PUSHDSILENT`

If this variable is defined, **gsh** will not print the directory stack after any of the directory stack commands (See Chapter 4 **pushd**, **popd**, .)

`$SAVEHIST`

A numeric value that contains the number of commands to save to disk when exiting **gsh**. These commands are then read back in when **gsh** is restarted which allows old commands to be reused. If the value is 0 or `SAVEHIST` is undefined, no commands will be saved to disk.

`$TERM`

This variable contains the name of the terminal emulation that the shell and other applications should use. By default, it is 'gnocron'. When the shell encounters a 'set term' command, it automatically calls the **tset** command to reload the termcap information. See Appendix F **Termcaps**.

`$TERMCAP`

This variable specifies the location of the *termcap* file. The shell and other applications look for *termcap* in the /etc directory, but if `TERMCAP` is set, the fully specified termcap file is used instead. This allows users to install custom termcap entries. See Appendix F **Termcaps**.

`$USER`

A string that represents the name of the current user. This variable is usually set by **login(1)**.

Accessing shell variables

Shell variables are defined or changed with the **set** command. The **unset** command is used to delete a variable. See Chapter 4 **Environment Commands** for more information on the **set** and **unset** commands

To access shell variables from the command line or a shell script, use the character "\$" followed by the variable name. The dollar sign character will expand the variable to its value. If you wish to use the dollar sign character in a string from the command line, remember to enclose it in single quotes or use the "\" escape character. If you use double quotes, the shell will try to expand the variable. To differentiate the variable name from characters that may immediately follow it, the variable name may be optionally surrounded with braces, "{" and "}". This provides a very powerful way of user interaction with shell scripts. As an example,


```
[1] 9:41pm root> set
PAGER = less
SAVEHIST = 25
PATH = :hard:gno:bin :right:gno:usr:bin
USRMAN = /usr/man
HISTORY = 100
status = 0
figignore = .a .root .sym
TERM = gnocon
LESS = -e
PROMPT = [%h] %S%t%s %C>
HOME = :hard:gno:user:root
user = root
nonewline = 1
[2] 9:41pm root> set history 30 user user1 ; set
PAGER = less
SAVEHIST = 25
PATH = :hard:gno:bin :right:gno:usr:bin
USRMAN = /usr/man
HISTORY = 30
status = 0
figignore = .a .root .sym
TERM = gnocon
LESS = -e
PROMPT = [%h] %S%t%s %C>
HOME = :hard:gno:user:root
user = user1
nonewline = 1
[3] 9:41pm root> unset savehist ; set
PAGER = less
PATH = :hard:gno:bin :hard:gno:usr:bin
USRMAN = /usr/man
HISTORY = 30
status = 0
figignore = .a .root .sym
TERM = gnocon
LESS = -e
PROMPT = [%h] %S%t%s %C>
HOME = :hard:gno:user:root
user = user1
nonewline = 1
[6] 9:43pm root> echo 'Current value of $path = ' $path
Current value of $path = :right:gno:bin :right:gno:usr:bin
[7] 9:44pm root> echo "value of $path = " $path
value of :right:gno:bin :right:gno:usr:bin = :right:gno:bin
:right:gno:usr:bin
[9] 9:45pm root> set user1 foo
[10] 9:45pm root> echo $user1
foo
[11] 9:45pm root> echo ${user}1
user11
[12] 9:46pm root> echo "echo '\$path = ' \$path" >> change.path
[13] 9:46pm root> echo "set path \$1" >> change.path
[14] 9:47pm root> echo 'rehash' >> change.path
[15] 9:47pm root> cat change.path
echo '$path = ' $path
set path $1
rehash
[16] 9:48pm root> change.path '/dev/gno/utilities'
```



```
$path = /dev/gno/orca /dev/gno/utilities
hashed 22 files.
[17] 9:49pm root> change.path $PATH
$path = /dev/gno/orca /dev/gno/utilities
hashed 55 files.
[18] 9:50pm root> echo this is $homeplate
this is                                     { variable homeplate wasn't found }
[19] 9:51pm root> echo this is ${home}plate
this is /dev/gno/plate                     { variable home is found }
```


Appendix A

Sample gsh session

The following represents a sample session in gsh. Many of the features discussed in this manual are reviewed here. Comments are enclosed in braces ("{}").

```
[1] gno> set
savehist = 25
path = /dev/orca/utilities /dev/gno/orca /dev/gno/utilities
history = 25
prompt = [%h] %C>
user = achin
home = /dev/gno
nonewline = 1
[2] gno> echo environment variable PROMPT = $PROMPT
environment variable PROMPT = [%h] %C>
[3] gno> cat gshrc
```

{ gshrc is the startup file. }

```
###
#
# GNO 2.0 gshrc file
#
###
#
# Initialize our environment
#
set path=":hard:gno:bin :right:gno:usr:bin"
set prompt="[%h] %S%t%s %C> "
set home=":hard:gno:user:root"
set term=gnocon
export path prompt home term
setenv history=100 savehist=25
###
#
#Set up standard prefixes for utilities.
#
###
prefix 2 :software:orca:libraries
prefix 3 :software:orca
prefix 4 :software:orca:shell
prefix 5 :software:orca:languages
prefix 6 :software:orca:utilities
prefix 7 :tmp
###
#
# Set up prefixes for Orca2.0(tm)'s benefit
#
###
prefix 13 :software:orca:libraries
prefix 14 :software:orca
prefix 15 :software:orca:shell
prefix 16 :software:orca:languages
prefix 17 :software:orca:utilities
alias ls 'ls -CF'
alias dir 'ls -al'
```



```

alias cp 'cp -i'
alias rm 'cp -p rm'
alias mv 'cp -p mv'
setenv usrman='/usr/man'
set fignore='.a .root .sym'
alias zcat 'compress -cd'
setenv pager=less
setenv less=-e
set nonewline=1
#
# Move to home directory
#
cd
[4] gno> cat gshrc > .ttyb&
                                     { .ttyb is the printer connected to the printer port. }

[2]
[5] gno> echo File \'gshrc\' sent to printer.
File 'gshrc' sent to printer.
[6] gno> alias l 'ls -F'; l
                                     { The ls "-F" option displays executable and directory files differently. }

:dev:gno
doc/          foo*          gsh*          gshrc*        kern*
man/          message      message2      orca/         press.release
release.notes samples/      todo         updates       utilities/

[7] gno> history
1: set
2: echo environment variable PROMPT = $PROMPT
3: cat .gshrc
4: cat .gshrc > .printer&
5: echo File \'gshrc\' sent to printer.
6: ls
7: history
[8] gno> no_file
no_file: Command not found.
[9] gno> message
message: Not executable.
[10] gno> unset USER
                                     { Environment variables are not case sensitive. }

[11] gno> set PATH

path = /dev/orca/utilities /dev/gno/orca /dev/gno/utilities
[12] gno> set PROMPT

prompt = [%h] %C>
[13] gno> unset PROMPT
% cp message message2&
[2]
% rm doc&
rm: :dev:gno:doc directory
% rm -r doc&
[3]
% ls *
:dev:gno
foo*          gsh*          gshrc*        kern*        man/
message      message2      orca/         press.release release.notes
samples/     todo         updates       utilities/
% echo *
gshrc foo message Finder.Data utilities press.release man release.notes todo
updates gsh kern samples doc orca message2

```



```
% echo g* m*
gshrc gsh message man message2
% echo g?
No match.
% pwd
/dev/gno
% clear ; df
```

.d##	Volume	Device	Free	Total	Capacity	System
.d1	:system	.APPLESCSI.HD01.	25746	50773	49%	ProDOS
.d2	:dev	.APPLESCSI.HD01.	44868	65108	31%	ProDOS
.d3	:usr	.APPLESCSI.HD01.	44526	48725	8%	ProDOS
.d4	:ram5	.DEV2	493	512	3%	ProDOS

```
% which ls cp echo ps
ls: /dev/gno/utilities/ls
cp: /dev/gno/utilities/cp
echo: Shell Built-in Command
ps: Shell Built-in Command
% echo -n foo
foo % 1 .dev2 .applescsi.hd01.01
      { Because of the previous "echo -n", the PROMPT is displayed on the line where echo output ended. }

:ram5
df          echo          history          ls              ls.ram5
ls.system   prefix         ps              set             which

:system
AppleworksGS/ BASIC.System* HyperCard/ ProDOS* a2fx/
basic.launcher* emacs.icon fontasm/ gutenber/ icons/
kermit/ publish.it/ review shr.pic/ shrinkit/
snowterm/ system/

% alias print 'echo'; alias copy 'cp'; alias delete 'rm'; alias
catalog 'ls -l'; alias rename 'mv'
      { Useful aliases for Orca fans. }

% alias
1: ls -F
print: echo
copy: cp
delete: rm
catalog: ls -l
rename: mv
% set PROMPT='[!] %C> '
[30] gno> cd samples
[31] samples> cml ps.c keep=ps > .dev2/ps&
[2]
[32] samples> cml kill.c keep=kill > .dev2/kill&
[3]
[34] samples> ls dummy_dir
[35] samples> ls -9
ls: unrecognized option '-9'
usage: ls [-acdfgilqrstulACLFR] [name ...]
[36] samples> ls -9 >& .dev2/error
      { Each pathname has a device name entry (cf. ps). }
[37] samples> cat < /ram5/error
ls: unrecognized option '-9'
usage: ls [-acdfgilqrstulACLFR] [name ...]
[38] samples> ls *.c;grep \#i\n\c\l\u\d\o [^sfd]*.c
      { grep searches for a regular expression in a file and outputs the lines that match. }

comp.c      dp.c      edit.c      ffdir.c      gar.c
kill.c      link.c     ps.c        setdebug.c
```



```

kill.c:#include <stdio.h>
kill.c:#include <memory.h>
kill.c:#include "3/work/gno/conf.h"
kill.c:#include "3/work/gno/proc.h"
kill.c:#include "3/work/gno/kerntool.h"
gar.c:#include <stdio.h>
gar.c:#include <orca.h>
gar.c:#include "3/work/gno/kerntool.h"
gar.c:#include <gno/signal.h>
ps.c:#include <stdio.h>
ps.c:#include <memory.h>
ps.c:#include <string.h>
ps.c:#include <orca.h>
ps.c:#include "3/work/gno/conf.h"
ps.c:#include "3/work/gno/proc.h"
ps.c:#include "3/work/gno/kvm.h"
edit.c:#include <stdio.h>
edit.c:#include <stdlib.h>
edit.c:#include <string.h>
edit.c:#include <ctype.h>
edit.c:#include <types.h>
edit.c:#include <gsos.h>
edit.c:#include <shell.h>
edit.c:#include <orca.h>
edit.c:#include <stddef.h>
edit.c:#include <texttool.h>
edit.c:#include <gno/kerntool.h>
comp.c:#include <stdio.h>
comp.c:#include <stdlib.h>
comp.c:#include <string.h>
comp.c:#include <ctype.h>
comp.c:#include <types.h>
comp.c:#include <shell.h>
comp.c:#include <gsos.h>
comp.c:#include <orca.h>
comp.c:#include <texttool.h>
comp.c:#include <gno/kerntool.h>
link.c:#include <stdio.h>
link.c:#include <stdlib.h>
link.c:#include <string.h>
link.c:#include <ctype.h>
link.c:#include <types.h>
link.c:#include <shell.h>
link.c:#include <orca.h>
link.c:#include <texttool.h>
link.c:#include <gno/kerntool.h>

```

```
[39] samples> head link.c
```

{ head displays the first few lines of a file. }

```
/*
```

parsing code for compile, cmpl, cmplg

The options flags are set up in a bizarre format as follows:

```

76543210 76543210 76543210 76543210
yz      qrstuvwxyz ijklmnop abcdefgh
[40] samples> set PATH
path = /dev/orca/utilities:/dev/gno/orca:/dev/gno/utilities
[41] samples> unset PATH

```



```
[42] samples> set PATH
Variable Not Found
[43] samples> rm m[e][s][s]age[2]
[44] samples> rehash
hashed 0 files.
[45] samples> rm who_cares about_the parameters
{ rm will not be recognized as a command because the hash-table is empty. }
rm: Command not found.
[46] samples> set path = /dev/orca/utilities ~/orca ~/utilities

/dev/orca/utilities /dev/gno/orca /dev/gno/utilities =
[47] samples> set PATH

path = =
[48] samples> set path="~/../orca/utilities ~/orca $HOME/utilities"
{ Filename expansion will be performed before PATH is set. }
[49] samples> set path

path = /dev/orca/utilities /dev/gno/orca /dev/gno/utilities
[49] samples> rehash
hashed 68 files.
```


Appendix B

Prefix Conventions

When **gsh** is started, GS/OS assigns certain values to individual prefixes, and usually the *gshrc* file also sets some prefixes. A total of 32 prefixes are available to the user. The following list documents each prefix and the purpose of each.

<u>Number</u>	<u>Description</u>
@	AppleShare prefix. If GNO resides on an AppleShare volume, this prefix is set to the pathname of the user's directory on the file server; otherwise, this prefix is set to the same pathname as prefix number 9.
*	Boot volume prefix. It is not possible to modify the value of this prefix with the shell's <code>prefix</code> command. The only other way to access this prefix is the GS/OS <code>_GetBootVol</code> call.
0	Prefix 0 is the current working directory. It is the prefix that is changed by the <code>cd</code> command.
1	Directory a program resides in. In the shell, this is usually <code>/bin</code> . The kernel sets this prefix (and 9/) appropriately for each program that is executed.
2-8	Used by Orca 1.0 languages and utilities
2	If the ORCA languages are being used, prefix 2 points to the ORCA Libraries directory.
3	This prefix should point to the directory that contains ORCA.SYS16.
4	ORCA Shell directory should point to the directory that contains the EDITOR, SYSTABS, SYSCMND, etc. files.
5	ORCA Languages directory
6	ORCA Utilities directory
7	ORCA Temp directory - setting this to a RAM disk makes certain editor operations faster.
9	Same as prefix 1/
10-12	Standard input, output, and error device names. GNO sets these to be <code>'.ttyco'</code> .
13	Same as prefix 2/ above if Orca 2.0 languages are being used.
14	Same as prefix 3/
15	Same as prefix 4/
16	Same as prefix 5/
17	Same as prefix 6/
18	Same as prefix 7/

Appendix C

Errors

gsh tries, when an error occurs, to output an informative error message that will lead you to the solution of your problem. This appendix documents all **gsh** error messages and what the probable cause of the problem might be. There are five classes of errors: generic **gsh**, command-entry, syntax, execution, and built-in. Each error is discussed separately.

Generic **gsh** Errors

These errors can typically occur at any time and may not be directly related to something the user has done. Some of them are trivial, and some are very serious and should be reported immediately.

gsh: There are stopped jobs.

All jobs must be killed before exiting the shell. Use the **jobs** and **kill** commands.

Command Editing Errors

Command editing errors occur when entering information on the command-line. If you try to move the cursor too far to the left or right of your command-line (i.e. before the first character or after the last character), an error will occur. At present, **gsh** indicates a command-entry error with a **Beep**. This is to notify you that the action you requested is not possible.

Syntax Errors

Syntax errors occur while **gsh** is trying to understand the command you have entered on the command-line. Problems arise when you wish to quote an argument (") and only enter one quote.

gsh: Missing ending ".

A second " wasn't supplied when quoting text.

gsh: Missing ending '.

A second ' wasn't supplied when quoting text.

gsh: Too many arguments, so no dessert tonight.

The command-line contained too many arguments which exceeded the available memory allocated by **gsh**.

gsh: Not enough memory for arguments.

No memory was available for allocating command-line arguments.

gsh: Extra '<' encountered.

gsh: Extra '>' or '>>' encountered.

gsh: Extra '>&' or '>>&' encountered

Text may be redirected to only one file.

gsh: No file specified for '<'.

gsh: No file specified for '>' or '>>'.

gsh: No file specified for '>&' or '>>&'.

A file must be specified when redirecting I/O.

gsh: '/' conflicts with '>' or '>>'.

gsh: '/' conflicts with '<'.

Piping is another form of redirection, thus pipes and redirections cannot be mixed.

Execution Errors

After **gsh** parses the command-line, it will then execute the command and pass any arguments to the command. If, however, the command does not exist, **gsh** will report an error. The reason the command does not exist could be either the command name was typed wrong or the command does not exist.

\$0: Command not found.

\$0 represents the command to be executed. Either the command name was entered incorrectly or the command does not exist. Recheck the spelling of the command and check **\$PATH** to make sure the command exists in the pathname list.

\$0: Not executable.

\$0 represents the command to be executed.

heh heh, next time you'll need to specify a command before redirecting.
Redirection was specified but the command-line had no command.

Cannot fork (too many processes?)

An error was encountered forking a process. The most likely culprit is there are too many processes running.

Built-in Command Errors

These are errors which can be returned by many of the built-in commands. Every built-in also contains a Usage message on the proper invocation method.

cd: Not a directory

Tried to change the cwd to a file that isn't a directory.

prefix: could not set prefix, pathname may not exist.

GS/OS Prefix command failed, most likely the pathname did not exist or the disk is damaged.

setdebug: Unknown flag

An unknown flag was sent to **setdebug**. Run **setdebug** with no arguments for a list of possible flags.

ps: error in kvm_open()

ps was unable to access the process data structure. It would be amazing that the kernel is still running for this error to occur.

set: Variable not specified

A variable was not passed to **set**, for example, "**set =bar**". Make sure the variable name was specified without the preceding dollar sign. For example, if **foo** is not set, then "**set \$foo=bar**" would be expanded to "**set =bar**", this resulting in this error.

kill: Invalid signal number

kill: Invalid signal name

See Appendix C for a list of valid signal numbers and names.

fg: No job to foreground.
bg: No job to background.
stop: No job to stop.
There aren't any jobs so this command is useless.

fg: No such job.
bg: No such job.
stop: No such job.
kill: No such job.
The specified job (or process) doesn't exist.

fg: Gee, this job is already in the foreground.
bg: Gee, this job is already in the background.
stop: Gee, this job is already stopped.

Well, this should be self-explanatory. Also, some of these should be impossible to get, unless you're bound and determined to crash gsh, but then, these errors will keep you from crashing it, so, what's the point?

Appendix D

Signals

The following list describes all signals present in the GNO Multitasking Environment. These signals are used to communicate with processes and the GNO kernel. **gsh** provides a means to send signals to processes via the kill command. Furthermore, GNO/ME provides interfaces for C and Assembly Language programmers to handle signals in their programs. Note that not all of these signals are actually used in GNO/ME 2.0.

<u>Name (Value)</u>	<u>Description</u>
SIGHUP (1)	Hangup.
SIGINT (2)	Interrupt.
SIGQUIT (3)	Quit.
SIGILL (4)	Illegal Instruction.
SIGTRAP (5)	Trace trap.
SIGABRT (6)	Abort (cf. abort(3)).
SIGEMT (7)	Emulator trap.
SIGFPE (8)	Arithmetic exception.
SIGKILL (9)	Kill. This signal cannot be caught, blocked, or ignored. If a signal is not specified for the kill command, this signal is sent to the process.
SIGBUS (10)	Bus error.
SIGSEGV (11)	Segmentation violation.
SIGSYS (12)	Bad argument to system call.
SIGPIPE (13)	Write on a pipe or other socket with no one to read it.
SIGALRM (14)	Alarm clock.
SIGTERM (15)	Software termination signal.
SIGURG (16)	Urgent condition present on socket.
SIGSTOP (17)	Stop. This signal cannot be caught, blocked, or ignored.
SIGTSTP (18)	Stop signal generated from keyboard.
SIGCONT (19)	Continue after stop. This signal cannot be blocked.
SIGCHLD (20)	Child status has changed.
SIGTTIN (21)	Background read attempted from control terminal.
SIGTTOU (22)	Background write attempted to control terminal.
SIGIO (23)	Input/Output possible signal.
SIGXCPU (24)	Exceeded CPU time limit.
SIGUSR1 (30)	User defined signal 1.
SIGUSR2 (31)	User defined signal 2.

Appendix E

Non-Compliant Applications

GNO/ME wasn't really designed with the intention of making EVERY program you currently run work under GNO/ME; that task would have been impossible. Our main goal was to provide a UNIX-based multitasking environment; that we have done. We made sure as many existing applications as we had time to track and debug worked with GNO/ME. The current list of compatible and non-compatible applications can be found in the file "RELEASE.NOTES" on the GNO/ME disk.

However, due to the sheer number of applications and authors, there are some programs that just plain don't work; and some that mostly work, except for annoyances such as two cursors appearing, or keyboard characters getting 'lost'. The problem here is that some programs use their own text drivers (since TextTools output was very slow at one time); since GNO/ME doesn't know about these custom drivers, it goes on buffering keyboard characters and displaying the cursor. There is a way, however, to tell GNO/ME about these programs that break GNO/ME's rules.

We've defined an auxType for S16 and EXE files, to allow distinction between programs that are GNO/ME compliant and those that are not. Setting the auxType of an application to \$DC00 disables the interrupt driven keyboard buffering and turns off the GNO/ME cursor. Desktop programs use the GNO/ME keyboard I/O via the Event Manager, and thus should *not* have their auxType changed.

You can change a program's auxType with the following shell command:

```
chtyp -a \ $DC00 filename
```

where filename is the name of the application. As more programmers become aware of GNO/ME and work to make their software compatible with it, this will become less of a problem, but for older applications that are unlikely to ever change (like the America OnLine software) \$DC00 is a reasonable approach.

Appendix F

Termcaps

'Termcap' is short for 'terminal capability', and is the name of a database which applications can use to do full-screen output on any kind of terminal. The termcap database contains records for the various supported terminals, each of which contains fields that look like this:

cap=value

cap is a two-letter code that represents a cursor movement, screen mode change (such as inverse or underline mode), and various other things. **Value** is usually a sequence of control characters that is sent to a terminal to initiate the desired action. **Value** can also be 'boolean', or yes/no, values, for such things as "Does this terminal support cursor movement?". The termcap file is documented in the electronic manual entry **termcap(5)**.

The termcap library does not specifically require GNO/ME.

The following terminal types are supported in the GNO/ME termcap file:

gnocon	GNO Console
CONSOLE	GS/OS .console driver
ptse	Proterm Special Emulation
vt100	DEC VT-100 terminal
ansisys	MS-DOS ANSI.SYS
xerox820	Xerox 820-II CP/M terminal
iw1	Apple ImageWriter I printer
iw-alt	Alternate ImageWriter I printer
deskjet	Hewlett Packard DeskJet 500 printer

The printer entries allow a formatted electronic manual page to be sent to the printer. For example, the following script would bring up the manual page for 'ls', format it for the DeskJet 500, and print it with italics and boldface:

```
set temp=$term
set term=deskjet
man $1 > .ttyb
set term=$temp
```


Glossary

Alias.	A name used as an abbreviation for one or more commands. An alias allows you to replace any command string with a short sequence of characters.
Applesoft.	An implementation of BASIC for the Apple II.
APW.	Apple Programmer's Workshop. Similar to ORCA.
BASIC.	Beginners All-purpose Symbolic Instruction Code. A simple computer language.
Built-in command.	A command processed by gsh . These commands are not external to the shell, but are included within the gsh program.
Command.	An action for gsh to perform. Commands can be either simple or compound. A simple command is an alias assignment, variable assignment, I/O redirection, or built-in command. A compound command is a pipeline.
Directory.	A special type of file that contains a list of other files; usually used to categorize files related in some way.
Environment.	The state of a process, which includes information such as its open files, current directory (working directory), and local and global variables. Three environments exist under gsh : <i>Child environment</i> :: The environment of the child process. <i>Current environment</i> :: The environment of the current process. <i>Parent environment</i> :: The environment of the parent process.
Environment file.	A file that is interpreted by an application to allow the user to customize its operation. For gsh , this file is "gshrc".
Export.	A way to pass a variable from a parent process to child process.
File.	An object used to store data and/or programs. On the IIGS, files are tagged with types such as EXE, SRC, TXT, etc.
Filter.	A command that reads from its standard input and writes to its standard output. For example, a filter program could be written to convert all characters to upper case. Filters are used mainly in pipelines.
Flag.	A character used to represent an option to a command. Flags are either short or long options whose character representations are "-" and "+".
Glob.	Slang for Pathname Expansion.
GNO/ME.	GNO Multitasking Environment. The complete package including the GNO kernel and the GNO Shell.
GNO Kernel.	Heart of GNO/ME. Executes processes when asked by the GNO Shell
GNO Shell.	Provides an interface between the user and the GNO kernel.
gsh.	GNO Implementation of a UNIX-like shell.
GS/OS.	16 bit Operating System for the Apple IIGs.
History.	A variable number of command-lines saved by gsh for future reference. The number of command-lines saved is dependent on the \$HISTORY environment variable.
History file.	A file containing command-lines entered while in a gsh session. The number of command-lines saved is dependent on the \$SAVEHIST environment variable.
Interrupt.	A signal generated by a sequence of keyboard characters or by a command that terminates the current executing process, unless the process has set up a trap to handle the interrupt signal.
I/O Redirection.	The process of changing the standard input, standard output, and standard error associated with a process so that it is redirected to a file instead of the console.
Job.	A set of related processes. A job can be either: <i>Background job</i> :A process that executes with the current process. Background jobs are not associated with the terminal.

	<i>Foreground job</i> : A process that is currently executing and which is associated with the terminal.
<i>Multiprocessing.</i>	Indicates a machine with more than one CPU.
<i>Multitasking.</i>	The ability to run more than one program at a time, or the illusion of more than one program running at a time - usually the latter.
<i>ORCA</i>	Shell programing environment for the Apple //gs. Also a type of whale.
<i>Path search.</i>	The means of searching a pathname list for a command or script.
<i>Pathname.</i>	A string used to identify a file.
	<i>Pathname completion.</i> The means of generating all pathnames matching a given pattern.
	<i>Pathname expansion.</i> The means of replacing a pattern with a list of pathnames matching that pattern.
<i>Pattern.</i>	A string of characters used to match literal characters and/or multiple characters.
<i>Permission.</i>	Each file has certain permissions associated with it: destroy, rename, backup, invisible, write, and read.
<i>Pipe.</i>	A conduit through which a stream of characters can pass from one process to another. This is accomplished by linking the standard output of one process to the standard input of a second process.
<i>Pipeline.</i>	Two or more processes connected together by pipes.
<i>Process.</i>	A single thread of execution that consists of a program and an execution environment.
	<i>Child process:</i> A new process created by another process.
	<i>Parent process:</i> A process that creates a child process.
<i>Process id:</i>	Each active process is uniquely identified by a positive integer called the process id.
<i>ProDOS.</i>	8-bit Disk Operating System for Apple II computers.
<i>Prompt.</i>	A message displayed by gsh when it is ready to receive a command.
<i>Quoting.</i>	A means of including special characters as arguments to a command or as the command name. Certain characters have certain meanings to gsh and quoting them makes gsh ignore them.
<i>Reserved word.</i>	A word that is treated specially by gsh . This word is part of the gsh grammar.
<i>Script.</i>	A sequence of commands contained in a file.
<i>Signal.</i>	An asynchronous message that consists of a number or name that can be sent from one process to another.
<i>Standard Error.</i>	The file associated with error messages for a process. This file is usually the terminal.
<i>Standard Input.</i>	The file associated with a processes input. This file is usually the terminal.
<i>Standard Output.</i>	The file associated with a processes output. This file is usually the terminal.
<i>Tilde Expansion.</i>	Words beginning with "~" are treated specially by gsh . The "~" is expanded to the value of the HOME variable.
<i>UNIX.</i>	Popular operating system which has growing use in education and business. One of the first operating systems to support multitasking.
<i>Variable.</i>	A named location in gsh that contains text. The text of a variable can be expanded in a command by preceding the variable name with a dollar sign (\$).
<i>Wildcard.</i>	See Pattern and Pathname Expansion.
<i>Working directory.</i>	The current directory.

Index

^C 19
\$ 35
^Z 19
\$0 33
\$1 33
\$2 33
\$< 33
\$DC00 50
\$ECHO 34
\$FIGIGNORE 10, 34
\$HISTORY 9, 34
\$HOME 9, 34
\$IGNOREEOF 34
\$NOBEEP 34
\$NODIREXEC 34
\$NOGLOB 34
\$NONEWLINE 34
\$PATH 31, 34
\$PRECMD 34
\$PROMPT 34
\$PUSHDSILENT 35
\$SAVEHIST 9, 35
\$TERM 35
\$TERMCAP 35
\$USER 35
& 17
. 21
.. 21
.i.OA 26
< 16
> 16
>& 16
>> 16
>>& 16
\n 27, 35
\r 27, 35
\t 27, 35
accelerator 19
alias 15, 30, 34
ansisys 51
AppleShare 45
argument 22
auxType 49
Background 17, 19
backward-char 8, 25
backward-delete-char 8, 25
backward-word 9, 25
BASIC 13
beginning-of-line 9, 25, 26
bg 19, 29

bindkey 7, 25
BLOCKED 29
Bourne shell 3
Built-In Commands 7
built-ins 25
C shell 3
cat 16, 22
cd 26, 45
chdir 26
child process 33
chtyp 50
CLEAR 8, 26
clear-screen 8, 25
cmpl 18, 19
coff 14, 17
command arguments 7
command name 7, 15
command-line 3, 7, 11, 15, 17, 19
command-line buffer 15
command-line editor 7, 9, 10, 25, 26
commands 26
complete-word 10, 25
CONSOLE 51
cp 18, 20, 22, 33
CTRL-A 9, 26
CTRL-B 8
CTRL-D 8, 9, 10
CTRL-E 9
CTRL-F 8
CTRL-K 26
CTRL-L 8
CTRL-N 10
CTRL-P 10
CTRL-R 8
CTRL-X 8
DELETE 8
delete-char 9, 25
deskjet 51
Desktop 50
df 26
dirs 28
done 19
DOWN-ARROW 10
down-history 10, 25
echo 7, 16, 24, 27
~ 21
editor 4, 5
end-of-line 9, 25
Errors 46
ESC 11, 26
ESC-E 11
ESC-Y 26
Event Manager 50

EXE 5, 7, 25, 33, 34, 50
EXEC 5, 7
exit 27
export 33
fg 19, 29
filter processes 17
foreground 19
forward-char 8, 25
forward-word 9, 25
globbing 22, 34
GNO Installer 3, 4, 9
GNO Kernel 3, 4, 5, 17, 25, 49
GNO shell 3, 4, 5, 9, 15, 17
GNO/ME 3, 4, 5, 13, 16, 17, 19, 49, 50
gnocon 35, 50
grep 17, 22
GS/OS 16
gsh 3, 5, 7, 9, 10, 11, 15, 16, 18, 19, 21,
22, 24, 26, 33, 49
gshrc 3, 4, 5, 9, 11, 15, 33
GUI 3
hash 31
HFS 23, 34
HISTORY 9, 27, 34
history buffer 9
home directory 21
I/O 50
insert 9, 11
iw-alt 51
iwl 51
job 19
Job Control 19
job number 17
jobs 29
kern 5
kernel 18, 26
kill 19, 29
kill-end-of-line 9, 25
kill-whole-line 8, 25
Korn shell 3
LEFT-ARROW 8
less 17
list-choices 10, 25
login(1) 35
long options 13, 14
ls 13, 14, 16, 18, 19, 22, 33, 51
man 13, 14, 17
manual pages 13
mkdir 22
modem 16
more 13, 17, 19, 23
MouseText 33
multitasking 17, 19, 33
mv 33
NEW 29
newline 8, 25
NullProcess 18
OA 7, 11, 26
OA-< 9, 26
OA-> 9
OA-CLEAR 26
OA-D 9
OA-E 9, 11
OA-LEFT ARROW 9
OA-RIGHT-ARROW 9
OA-Y 26
OMF 17
Open Apple 7
option arguments 13
ORCA 4, 5, 15, 16
ORCA shell 3, 17
overstrike 9
paging 17
parameters 7
parent process 33
PATH 4, 5, 24
path delimiter 4
Pathname Expansion 22
pathname separator 4
PAUSED 30
performance 19
pipeline 16
popd 28, 35
pre-emptive multitasking 17
prefix 31
Prefix Conventions 45
printer 16
process 19
process ID 17
ProDOS 13, 23
prompt 5
ps 18, 19, 20, 29
ptse 51
pushd 28, 35
pwd 28
qtime 7
quote 35
raw-char 25, 26
READY 29
redirect 16
redirection 16
redirection operators 16
redisplay 25
redraw 8
regular expressions 22
rehash 24, 31

RELEASE.NOTES 50
Remote Access 11
RETURN 8
RIGHT-ARROW 8
running 19, 29
S16 50
Sample gsh session 39
SAVEHIST 9
Scope 33
script 11, 24, 28
serial port 16
set 31, 35
setdebug 30
setenv 31
shell script 33
short options 13, 14
SIGABRT 49
SIGALRM 49
SIGBUS 49
SIGCHLD 49
SIGCONT 49
SIGEMT 49
SIGFPE 49
SIGHUP 49
SIGILL 49
SIGINT 49
SIGIO 49
SIGKILL 49
SIGNAL 25, 29
signals 49
SIGPIPE 49
SIGQUIT 49
SIGSEGV 49
SIGSTOP 49
SIGSYS 49
SIGTERM 49
SIGTRAP 49
SIGTSTP 49
SIGTTIN 49
SIGTTOU 49
SIGURG 49
SIGUSR1 49
SIGUSR2 49
SIGXCPU 49
source 5, 28
standard error 16, 18
standard input 16, 33
standard output 16, 18
stop 19, 30
stopped 19
storage device 16
SUSPENDED 30
SYS16 5, 7, 34

TAB 10
TERM 33
termcap 35, 50
terminal 11, 16, 50
terminal emulation 35
TextTools 50
tilde 21
toggle-cursor 9, 25
tset 28, 35
ttya 16
unalias 15, 31
undefined-char 25, 26
unhash 31
UNIX 5, 14, 34, 50
unset 35
UP-ARROW 10
up-history 10, 25
variables 3, 33, 35
vi 4
vt100 51
WAITING 30
WAITSIGCH 30
which 28
wildcards 22
word 7
xerox820 51

GNO Kernel Reference Manual

By Jawaid Bazyar
Edited by Andrew Roughan



University of California, Berkeley

Division of Biological Sciences

Department of Molecular and Cellular Biology

Table of Contents

Chapter 1	Introducing the GNO Kernel.....	3
Chapter 2	GNO/ME Compliance.....	5
	Detecting the GNO Environment.....	5
	Terminal I/O.....	5
	Stack Usage	6
	Disk I/O.....	6
	Non-Compliant Applications.....	7
Chapter 3	Modifications to GS/OS.....	9
	Mutual Exclusion in GS/OS and ToolBox calls.....	9
	Pathnames and Prefixes.....	9
	Named prefixes	10
	Open File Tracking	10
	Refnums and file descriptors.....	11
	GNO/ME Character Devices.....	11
	Restartability.....	12
	Miscellaneous	12
Chapter 4	Modifications to the ToolBox.....	13
	TextTools Replacement	13
	SysFailMgr (\$1503).....	15
	The Resource Manager.....	15
	The Control Panel.....	16
	QDStartup(\$0204).....	16
Chapter 5	Process Management.....	17
	Process Table.....	18
	Task Switching.....	19
Chapter 6	Interprocess Communication.....	21
	Semaphores.....	21
	Signals	22
	Pipes.....	24
	Messages.....	25
	Ports.....	25
	Pseudo-Terminals (PTYs).....	26
	Deadlock.....	28
Appendix A	Making System Calls.....	29
	System Call Interface.....	29
	System Call Error Codes.....	30
	System Panics.....	31
Appendix B	Miscellaneous Programming Issues.....	32
	Option Arguments	32
	Pathname Expansion	32
Glossary		33
Index		35

Chapter 1

Introducing the GNO Kernel

The GNO kernel is the heart of the GNO Multitasking Environment (GNO/ME). The GNO kernel provides a layer of communication between the shell (and shell-based programs) and the operating system, GS/OS. The kernel handles such things as multitasking, background processes, foreground processes and many other features that were not previously available on the Apple II GS. It is these features which make GNO/ME very powerful.

This reference manual is highly technical in nature and is provided to help programmers develop utilities for the GNO Multitasking Environment. The beginner has no need to read this manual and is certainly not expected to understand its contents. However, Chapter 5 **Process Management** and Chapter 6 **Interprocess Communication** provide a good background discussion for anyone who is interested in the internal workings of the kernel.

(We're sorry, but you did pay 1 cent extra for this blank page)

Chapter 2

GNO/ME Compliance

For a program to work effectively under GNO/ME, certain rules must be followed. Most of these rules boil down to one underlying concept - **never directly access features of the machine**. Always use GS/OS, the ToolBox, or GNO/ME to accomplish what you need. We have taken great care to provide the sorts of services you might need, such as checking for input without having to wait for it. GNO/ME compliance isn't just a matter of trying to make applications work well under the environment; it ensures that those applications stay compatible, no matter what changes the system goes through. Below are summarized the points you must consider when you're writing a GNO/ME compliant application.

Detecting the GNO Environment

If your application requires the GNO Kernel to be active (if it makes any kernel calls), you can make sure of this by making a **kernStatus** call at the beginning of your program. The call will return no error if the kernel is active, or it will return an error code of \$0001 (Tool locator - tool not found), in which case the value returned will be invalid. The call actually returns a 1 if no error occurs, but the value returned will be indeterminate if the kernel is not active, so you should only check for an error (the function **toolerror()** or the variable **_toolErr** in C, the value in the A register in assembly).

You can also determine the current version of the GNO Kernel by making the **kernVersion** call. The format of the version number returned is the same as the standard ToolBox calls. For example a return value of \$0201 indicates a version of 2.1.

kernStatus and **kernVersion** are defined in the `<gno/gno.h>` header file.

Terminal I/O

The Apple II has always been lacking in standardized methods for reading keyboard input and controlling the text screen. This problem was compounded when Apple stopped supporting the TextTools in favor of the GS/OS console driver. The console driver has a number of problems that prevent it from being a good solution under GNO/ME. There is high overhead involved in using it. It is generally accessed like a regular file, which means any I/O on it must filter through several layers before being handled. Even though in System 6.0.1 there is a provision for patching the low-level routines the special high-level user input features of the driver cannot be used over a modem or in a desktop program. And GS/OS must be called to access it, which means that while a console driver access is occurring, no other processes can execute. See Chapter 3 **Mutual Exclusion in GS/OS and ToolBox calls**.

GNO/ME ignores the GS/OS `' .CONSOLE '` driver and replaces the TextTools with a high-performance, very flexible generic terminal control system. GNO/ME directly supports the console (keyboard and screen), as well as the serial ports, as terminals. In order for a user program to take advantage of these features and to be GNO/ME compliant, you must do terminal I/O only through the TextTools, or through `stdin`, `stdout`, and `stderr` (refNums 1,2, and 3 initially) via GS/OS. By its very nature TextTools is slow, so we recommend using them only for small and simple tasks. Calls to the GS/OS console driver will not crash the system, but they will make other processes stop until the call is completed.

You must not get input directly from the keyboard latch (memory location \$E0C000), nor may you write directly to the screen memory. GNO/ME's terminal I/O system has been designed so you don't have to do either of these things. If you need to check for keyboard input without stopping your application, you can make the appropriate `ioctl(2)` call to do what you need.

In the future, GNO/ME may provide a GNO/ME-friendly version of the GS/OS .CONSOLE driver.

Stack Usage

Stack space is at a premium on the Apple IIgs. Process stacks can only be located in Bank 0 - a total of 64K. This theoretical limit doesn't apply, however, as GS/OS and other bits of system software reserve a large chunk of this without any way to reclaim it. There is approximately 48K of usable stack space. This space also has to be shared with direct page space for Tools and certain types of device drivers, however. For a program to be GNO compliant, stack usage analysis must be done and acted upon. Use of the stack should be minimized so that many processes can coexist peacefully. From experience we've found that 1K usually suffices for well-written C applications, and at a maximum 4K can be allocated.

Assembly language programs tend to be very efficient when it comes to use of the stack. The 4K provided by default to applications is usually more than enough for assembly language programs. C programs can use up tremendous amounts of stack space, especially if recursion is employed or string manipulation is done without concern for stack usage; however, even assembly programs can be written poorly and use a lot of stack space. Below are some hints to keep stack usage at a minimum.

- Avoid use of large local arrays and character strings. Instead, dynamically allocate large structures such as GS/OS strings with `malloc()` or the Memory Manager. Alternatively, you can designate such items as 'static', which causes the C compiler to allocate the space for the variable from main memory.
- Try not to use recursion unless absolutely necessary. All recursive functions can be rewritten using standard loops and creative programming. This is a good general programming rule because your program will run faster because setting up stack frames is expensive in terms of time and memory.
- ORCA/C 1.3 (and older) generates 8K of stack by default, in case the desktop is started up. Since GNO/ME compliant programs generally will not be desktop-based, make sure you judge how much stack your program will require and use the `#pragma stacksize` directive to limit how much stack space ORCA/C tries to allocate for your program. Also, since ORCA/C programs don't use the stack given them by GNO/ME and GS/OS, when you link your program include a small (256 bytes) stack segment. See the utilities sources for examples of this. ORCA/C 2.0 allocates stack via the GS/OS supported method, so ORCA/C 2.0 programs use exactly the amount of stack specified by `#pragma stacksize`.

Disk I/O

Since the Apple IIgs doesn't have coprocessors to manage disk access and the serial ports, either of these requires the complete attention of the main 65816 processor. This can wreak havoc in an environment with slow disks or high-speed serial links, as accessing disks usually results in turning off interrupts for the duration of the access. This situation is lessened considerably with a DMA disk controller, such as the Apple High Speed SCSI or CV Technologies RamFAST. But this isn't as bad as it sounds; the IBM PC and Apple Macintosh also suffer from this problem, and the solution is robust programming. Make sure

your communications protocol can handle errors where expected data doesn't arrive quite on time, or in full. The best solution would be an add-on card with serial ports and an on-board processor to make sure all serial data was received whether or not the main processor was busy (this is a hint to some enterprising hardware hacker, by the way).

Yet another concern for GNO/ME applications is file sharing. GS/OS provides support for file sharing, but it is up to the application author to use it via the requestAccess field in the OpenGS call. GS/OS only allows file sharing if all current references to a file (other instances of the file being opened) are read-only. GNO/ME authors should use read-only access as much as possible. For example, an editor doesn't need write permission when it's initially reading in a file. Note that the **fopen()** library routine in ORCA/C 1.2 does NOT support read-only mode (even if you open the file with a 'r' specifier), but it does in ORCA/C 1.3 and later.

Non-Compliant Applications

GNO/ME wasn't really designed with the intention of making EVERY program you currently run work under GNO/ME; that task would have been impossible. Our main goal was to provide a UNIX-based multitasking environment; that we have done. We made sure as many existing applications as we had time to track and debug worked with GNO/ME. The current list of compatible and non-compatible applications can be found in the file "RELEASE.NOTES" on the GNO/ME disk.

However, due to the sheer number of applications and authors, there are some programs that just plain don't work; and some that mostly work, except for annoyances such as two cursors appearing, or keyboard characters getting 'lost'. The problem here is that some programs use their own text drivers (since TextTools output was very slow at one time); since GNO/ME doesn't know about these custom drivers, it goes on buffering keyboard characters and displaying the cursor. There is a way, however, to tell GNO/ME about these programs that break GNO/ME's rules.

We've defined an auxType for S16 and EXE files, to allow distinction between programs that are GNO/ME compliant and those that are not. Setting the auxType of an application to \$DC00 disables the interrupt driven keyboard buffering and turns off the GNO/ME cursor. Desktop programs use the GNO/ME keyboard I/O via the Event Manager, and thus should *not* have their auxType changed.

You can change a program's auxType with the following shell command:

```
chtyp -a \ $DC00 filename
```

where filename is the name of the application. As more programmers become aware of GNO/ME and work to make their software compatible with it, this will become less of a problem, but for older applications that are unlikely to ever change (like the America OnLine software) \$DC00 is a reasonable approach.

Chapter 3

Modifications to GS/OS

The GNO system modifies the behavior of a number of GS/OS calls in order to allow many programs to execute concurrently, and to effect new features. The changes are done in such a way that old software can take advantage of these new features without modification. Following is a complete description of all the changes made. Each section has details in text, followed by a list of the specific GS/OS or ToolBox calls affected.

Mutual Exclusion in GS/OS and ToolBox calls

The Apple IIGS was not designed as a multitasking machine, and GS/OS and the Toolbox reflect this in their design. The most notable problem with making multitasking work on the Apple IIGS is the use of global (common to all processes) information, such as prefixes and direct page space for tool sets which includes information like SANE results, QuickDraw drawing information, etc. In most cases we've corrected these deficiencies by keeping track of such information on a per-process basis, that is, each process has its own copy of the information and changes to it do not affect any other process' information.

However, there were many other situations where this could not be done. Therefore, there is a limit of one process at a time inside either GS/OS or the Toolbox. GNO/ME automatically enforces this restriction whenever a tool or GS/OS call is made.

The method and details of making GS/OS calls does not change! The calls listed below have been expanded transparently. There are no new parameters and no new parameter values. In all cases, the corresponding ProDOS-16 interface calls are also supported, except ExpandPath and other calls which do not exist in ProDOS-16.

Pathnames and Prefixes

Normally under GS/OS there are 32 prefixes, and these are all under control of the current application. GNO/ME extends this concept to provide each process with its own copies of all prefixes. When a process modifies one of these prefixes via the GS/OS SetPrefix call, it modifies only its own copy of that prefix- the same numbered prefixes of any other processes are not modified.

Pathname processing has been expanded in GNO/ME. There are now two new special pathname operators that are accepted by any GS/OS call that takes a pathname parameter:

- . current working directory
- .. parent directory

For example, presume that the current working directory (prefix 0) is /foo/bar/moe. ". /ls" refers to the file "/foo/bar/moe/ls", and since a pathname was specified, this overrides the shell's hash table. "../ls" refers to "/foo/bar/ls". The operators can be combined, also, as in "... /ls" ("/foo/ls"), "... /ls" ("/foo/bar/ls"). As you can see, the '.' operator is simply removed and has no effect other than to force a full expansion of the pathname.

Shorthand device names (.d2, .d5, etc) as in ORCA are available only under System Software 6.0 and later. The common pathname operator '~' (meaning the home directory) is handled by the shell; if the character appears in a GS/OS call it is not treated specially.

\$2004 ChangePath	\$2006 GetFileInfo
\$200B ClearBackupBit	\$200A GetPrefix
\$2001 Create	\$2010 Open
\$2002 Destroy	\$2005 SetFileInfo
\$200E ExpandPath	\$2009 SetPrefix

Named prefixes

In order to allow easy installation and configuration of third-party software into all systems, GNO/ME provides a feature called named prefixes. These prefixes are defined in the /etc/namespace file. Basically, since all UNIX systems have /bin, /usr, /etc, and other similar standard partitions, but Apple IIgs systems generally do not have these partitions, named prefixes provide a way to simulate the UNIX directories without forcing GNO/ME users to rename their partitions (an arduous and problem-filled task).

Named prefixes are handled by the GNO kernel in the same GS/OS calls described in Chapter 3 **Pathnames and Prefixes**.

Open File Tracking

Previously, a major problem with the way GS/OS handled open files was that unrelated programs could affect each other's open files. For example, a Desk Accessory (or a background program of any sort) could open a file and have it closed without it's knowledge by the main application program. This presented all kinds of problems for desk accessory authors. Apple presented a partial solution with System Software 5.0.4, but it wasn't enough for a true multitasking environment. GNO/ME keeps track of exactly which process opened which file. It also discontinues the concept of a global File Level, opting instead for a per-process File Level. Any operations a process performs on a file (opening, closing, etc.) do not affect any other process' files.

In addition to this behavior, when a process terminates in any manner all files that it currently has opened will be closed automatically. This prevents problems of the sort where a program under development terminates abnormally, often leaving files open and formerly necessitating a reboot.

The Flush GS/OS call is not modified in this manner as its effects are basically harmless.

The Close call accepts a refNum parameter of 0 (zero), to close all open files. This works the same way under GNO/ME, except of course that only the files of the process calling Close are in fact closed.

\$2010 Open
\$2014 Close
\$201B GetLevel
\$201A SetLevel

Quitting applications

The QUIT and QuitGS calls have been modified to support the GNO/ME process scheme. Quitting to another application, whether by specifying a pathname or by popping the return stack, is accomplished with `execve(2)`. When there are no entries on the return stack, the process is simply killed. See the *GS/OS Reference Manual* for more details on how the Quit stack works.

Refnums and file descriptors

GS/OS tells you about open files in the form of refNums (reference numbers). UNIX's term for the same concept is 'file descriptor'. From a user's or programmer's view of GNO/ME, these terms are identical and will be used as such; which one depends on what seems most appropriate in context.

For each process, GNO/ME keeps track of which files that particular process has opened. No other process can directly access a file that another process opened (unless programmed explicitly), because it doesn't have access to any file descriptors other than its own. This is different from GS/OS in that GS/OS allows access to a file even if a program guessed the refNum, either deliberately or accidentally. This is one of the aspects of process protection in GNO/ME.

All of the various I/O mechanisms that GNO/ME supports (files, pipes, and TTYs) are handled with the same GS/OS calls you are familiar with. When you create a pipe, for example, you are returned file descriptors which, because of synonymity with refNums, you can use in GS/OS calls. Not all GS/OS calls that deal with files are applicable to a particular file descriptor; these are detailed in the sections on pipes and TTYs.

GNO/ME sets no limit on the number of files a process may have open at one time. (Most UNIX's have a set limit at 32).

GNO/ME Character Devices

GNO/ME supports a new range of character device drivers. These drivers are not installed like normal GS/OS drivers, but they are accessed the same way. There are the following built-in drivers:

- .TTYCO This is the GNO/ME console driver. The driver supports the TextTools Pascal control codes, plus a few GNO/ME-specific ones. These are documented in Chapter 4 **TextTools Replacement**. This driver is highly optimized both through the GS/OS and TextTools interfaces.
- .TTYA[0-9, A-F]
- .PTYQ[0-9, A-F] Pseudo-terminal devices; PTYs are used for interprocess communication and in network activities.
- .NULL This driver is a bit bucket. Any data written to it is ignored, and any attempt to read from it results in an end-of-file error (\$4C).

Just as with GS/OS devices, these GNO/ME drivers are accessed with the same Open, Read, Write, and Close calls that are used on files. Unlike GS/OS character devices, the characteristics of GNO/ME drivers are controlled through the `ioctl()` system call. The GS/OS Device calls (like DInfo, DStatus) are not applicable to GNO/ME drivers. See the `ioctl(2)` and `tty(4)` manpage for details.

Some GS/OS calls will return an error when given a refNum referring to a GNO/ME character driver or pipe because the concepts simply do not apply. The error returned will be \$58 (Not a Block Device), and the calls are as follows:

\$2016 SetMark	\$2017 GetMark
\$2018 SetEOF	\$2019 GetEOF
\$2015 Flush	\$201C GetDirEntry

GNO/ME loaded drivers (generally for serial communications, but other uses are possible) are configured in the `/etc/tty.config` file. Each line in `/etc/tty.config` describes one driver. The format of each line is:

```
filename      slot  devname
```

devname is the name of the device as it will be accessed (e.g. `.ttya`). **slot** is the slot in the device table where the device will be accessed from; it may refer to one of the physical expansion slots, as TextTools will use the specified driver when redirecting output to a slot. The **modem** and **printer** port drivers are configured for slots 2 and 1, respectively.

Pseudo-terminals are pre-configured into the kernel. PTYs are discussed further in Chapter 6 **Pseudo-terminals PTYs**.

Since `.ttyco` and the pseudo-terminals are pre-configured in the GNO kernel, entries for these devices do not appear in `/etc/tty.config`.

Restartability

GS/OS supports the concept of program 'restartability'. This allows programs which are written in a certain way to remain in memory in a purgeable state so that if they are invoked again, and their memory has not been purged, they can be restarted without any disk access. This greatly increases the speed with which restartable programs can be executed.

The ORCA environment specifies whether or not a program is restartable via a flag character in the SYSCMND file. The GS/OS standard method, however, is to set the appropriate flags bit in the GS/OS Quit call. This is the method that GNO/ME supports. Provided with the GNO/ME standard library is a routine **rexit(3)**. **rexit(3)** only works with ORCA/C 2.0. **rexit(3)** works just like the normal C **exit()** call but it sets the restart flag when calling QuitGS.

The standard ORCA/C 1.3 libraries are not restartable, but the ORCA/C 2.0 libraries are.

Miscellaneous

The following miscellaneous GS/OS calls have also been modified for GNO/ME.

\$2027 GetName	Returns the name on disk of the process. This only returns valid information after an execve(2) .
\$2003 OSShutdown	This call has been modified to kill all processes before performing the actual shutdown operation.

Chapter 4

Modifications to the ToolBox

Several changes have been made to the ToolBox, the most major of which is the replacement of the entire TextTools tool set. The TextTools were replaced for a number of reasons- better control over text I/O, increased speed, and emulation of ORCA's redirection system with as little overhead as possible. Other changes were made to modify the behavior of some tool calls to be more consistent with the idea of a multitasking environment.

TextTools Replacement

The changes to the TextTools have turned it into a much more powerful general I/O manager. The TextTools now intrinsically handle pipes and redirection, and you can install custom drivers for TextTools to use. Also, the TextTools have had their old slot-dependence removed; the parameter that used to refer to 'slot' in the original texttools calls now refers to a driver number. A summary of driver numbers (including those that come pre-installed into GNO) are as follows:

- 0 Null device driver
- 1 serial driver (for printer port compatibility)
- 2 serial driver (for modem port compatibility)
- 3 console driver (Pascal-compatible 80-column text screen)
- 4-5 User installed

See Chapter 3 **GNO/ME Character Devices**, for information on configuring these drivers.

There are also new device types in the TextTools; the complete list of supported device types and what their slotNum's (from SetInputDevice, SetOutputDevice, etc) mean is as follows:

- 0 Used to be BASIC text drivers. These are no longer supported under GNO/ME, and setting I/O to a basic driver actually selects a Pascal driver.
- 1 Pascal text driver. This is one of the drivers specified in /etc/ttys or built-in to GNO/ME.
slotNum: driver number as listed above
- 2 RAM-based Driver (documented in *ToolBox Reference Volume 2*)
slotNum: pointer to the RAM-based driver's jump table
- 3 file redirection
slotNum: refNum (file descriptor) of the file to access through TextTools

The new console driver supports all the features of the old 80-column Pascal firmware, and adds a few extensions, with one exception - the codes that switched between 40 and 80 columns modes are not supported. It is not compatible with the GS/OS 'console' driver. The control codes supported are as follows:

Hex	ASCII	Action
01	CTRL-A	set cursor to flashing block
02	CTRL-B	set cursor to flashing underscore
03	CTRL-C	Begin "Set Text Window" sequence
05	CTRL-E	Cursor on
06	CTRL-F	Cursor off

07	CTRL-G	Perform FlexBeep
08	CTRL-H	Move left one character
09	CTRL-I	Tab
0A	CTRL-J	Move down a line
0B	CTRL-K	Clear to EOP (end of screen)
0C	CTRL-L	Clear screen, home cursor
0D	CTRL-M	Move cursor to left edge of line
0E	CTRL-N	Normal text
0F	CTRL-O	Inverse text
11	CTRL-Q	Insert a blank line at the current cursor position
12	CTRL-R	Delete the line at the current cursor position.
15	CTRL-U	Move cursor right one character
16	CTRL-V	Scroll display down one line
17	CTRL-W	Scroll display up one line
18	CTRL-X	Normal text, mousetext off
19	CTRL-Y	Home cursor
1A	CTRL-Z	Clear entire line
1B	CTRL-[	MouseText on
1C	CTRL-\	Move cursor one character to the right
1D	CTRL-]	Clear to end of line
1E	CTRL-^	Goto XY
1F	CTRL-_ CTRL-	Move up one line

(Note: the *Apple IIgs Firmware Reference* incorrectly has codes 05 and 06 reversed. The codes listed here are correct for both GNO/ME and the Apple IIgs 80-column firmware)

FlexBeep is a custom beep routine that doesn't turn off interrupts for the duration of the noise as does the default Apple IIgs beep. This means that the beep could sound funny from time to time, but it allows other processes to keep running. We also added two control codes to control what kind of cursor is used. There are two types available as in most text-based software; they are underscore for 'insert' mode, and block for 'overstrike'. You may, of course, use whichever cursor you like. For example, a communications program won't have need of insert mode, so it can leave the choice up to the user.

The Set Text Window sequence (begun by a \$03 code) works as follows:

CTRL-C '[' LEFT RIGHT TOP BOTTOM

CTRL-C is of course hex \$03, and '[' is the open bracket character (\$5B). TOP, BOTTOM, LEFT, and RIGHT are single-byte ASCII values that represent the margin settings. Values for TOP and BOTTOM range from 0 to 23; LEFT and RIGHT range from 0 to 79. TOP must be numerically less than BOTTOM; LEFT must be less than RIGHT. Any impossible settings are ignored, and defaults are used instead. The extra '[' in the sequence helps prevent the screen from becoming confused in the event that random data is printed to the screen.

After a successful Set Text Window sequence, only the portion of the screen inside the 'window' will be accessible, and only the window will scroll; any text outside the window is not affected.

The cursor blinks at a rate defined by the **Control Panel/Options/Cursor Flash** setting. Far left is no blinking (solid), and far right is extremely fast blinking.

ReadLine (\$240C) now sports a complete line editor unlike the old TextTools version. Following is a list of the editor commands.

EOL	Terminates input (EOL is a parameter to the <code>_ReadLine</code> call)
LEFT-ARROW	Move cursor to the left
RIGHT-ARROW	Move cursor to right. Won't go past rightmost character.
DELETE	Delete the character to the left of the cursor.
CTRL-D	Delete character under the cursor.
OA-D	Delete character under the cursor.
OA-E	Toggles between overwrite and insert mode.

`ReadChar` (\$220C) has also been changed. The character returned may now contain the key modification flags (\$C025) in the upper byte and the character typed in the lower byte. This is still compatible with the old `TextTools` `ReadChar`. To get the `keyMod` flags, call `SetInGlobals` (\$090C) and set the upper byte of the AND mask to \$FF. Typical parameters for `SetInGlobals` to get this information are: `ANDmask = $FF7F`, `ORmask = $0000`.

The default I/O masks have also been changed. They are now `ANDmask = $00FF`, `ORmask = $0000`. They are set this way to extend the range of data that can be sent through `TextTools`. GNO/ME Character drivers do not, like the previous `TextTools` driver, require the hi-bit to be set.

The new `TextTools` are completely reentrant. This means that any number of processes may be executing `TextTools` calls at the same time, increasing system performance somewhat. The `TextTools` are also the only toolset which is not mutexed.

The GNO/ME console driver also supports flow-control in the form of Control-S and Control-Q. Control-S is used to stop screen output, and Control-Q is used to resume screen output.

SysFailMgr (\$1503)

The `MiscTool` call `SysFailMgr` has been modified so that a process calling it is simply killed, instead of causing system operation to stop. This was done because many programs use `SysFailMgr` when a simple error message would have sufficed. There are, however, some tool and GS/OS errors which are truly system failure messages, and these do cause system operation to stop. These errors are as follows:

\$0305	Damaged heartbeat queue detected
\$0308	Damaged heartbeat queue detected
\$0681	Event queue damaged
\$0682	Queue handle damaged
\$08FF	Unclaimed sound interrupt

What the system does after displaying the message is the same as for a system panic.

The Resource Manager

The Resource Manager has been modified in some subtle ways. First, GNO/ME makes sure that the `CurResourceApp` value is always correct before a process makes a Resource Manager call. Second, all open resource files are the property of the Kernel. When a `GetOpenFileRefnum` call is made, a new `refnum` is `dup()`'d to allow the process to access the file. Having the Kernel control resource files also allows all processes to share `SYS.RESOURCES` without requiring each process to explicitly open it.

The Event Manager

GNO/ME starts up the Event Manager so it is always available to the kernel and shell utilities. Changes were made so that the Event Manager obtains keystrokes from the GNO/ME console driver (.ttyco). This allows UNIX-style utilities and desktop applications to share the keyboard in a cooperative manner. This also makes it possible to suspend desktop applications; see Chapter 7, **Suspend NDA**.

EMStartUp sets the GNO console driver to RAW mode via an `ioctl()` call, to allow the Event Manager to get single keystrokes at a time, and to prevent users from being able to kill the desktop application with ^C or other interrupt characters. The four "GetEvent" routines, GetNextEvent, GetOSEvent, EventAvail, and OSEventAvail now poll the console for input characters instead of using an interrupt handler.

The Control Panel

In most cases, the CDA menu is executed as an interrupt handler. Since the Apple IIgs interrupt handler firmware isn't reentrant, task switching is not allowed to occur while the control panel is active. This basically means that all processes grind to a halt. In many ways, however, this is not undesirable. It definitely eases debugging, since a static system is much easier to deal with than a dynamic system. Also, CDAs assume they have full control of the text screen; multitasking CDAs would confuse and be confused in terms of output.

During the execution of the Control Panel, the original non-GNO/ME TextTools tool is reinstalled to prevent compatibility problems. Another step, taken to maintain user sanity, makes CDAs run under the kernel's process ID.

All the changes were made to two tool calls: SaveAll(\$0B05) and RestAll(\$0C05).

QDStartup(\$0204)

The QDStartup call has been modified to signal an error and terminate any process that tries to make the call when it's controlling terminal is not the Apple IIgs console. This prevents a user on a remote terminal from bringing up a desktop application on the console, an operation he could not escape from and one that would greatly annoy the user at the console.

Another change ensures that an attempt to execute two graphics-based applications concurrently will fail; the second process that tries to call QDStartUp is killed and a diagnostic message is displayed on the screen.

Chapter 5

Process Management

Before discussing process management using Kernel calls, it would be wise to define just exactly what we refer to when we say *process*. A process is generally considered to be a program in execution. "A program is a passive entity, while a process is an active entity." (Operating Systems Concepts p.73, Silberschatz and Peterson, Addison-Wesley, 1989). The concept of process includes the information a computer needs to execute a program (such as the program counter, register values, etc).

In order to execute multiple processes, the operating system (GNO/ME and GS/OS in this case) has to make decisions about which process to run and when. GNO/ME supports what is termed *preemptive multitasking*, which means that processes are interrupted after a certain amount of time (their time slice), at which point another process is allowed to run. The changing of machine registers to make the processor execute a different process is called a *context switch*, and the information the operating system needs to do this is called its *context*. The GNO kernel maintains a list of all active processes, and assigns time slices to each process according to their order in the list. When the kernel has run through all the processes, it starts again at the beginning of the list. This is called *round-robin scheduling*. Under certain circumstances, a process can actually execute longer than its allotted time slice because task switches are not allowed during a GS/OS or ToolBox call. In these cases, as soon as the system call is finished the process is interrupted.

Processes can give up the rest of their time slice voluntarily (but not necessarily explicitly) in a number of ways, terminal input being the most common. In this case, the rest of the time slice is allocated to the next process in line (to help smooth out scheduling). A process waiting on some event to happen is termed *blocked*. There are many ways this can happen, and each will be mentioned in its place.

An important item to remember is the *process ID*. This is a number which uniquely identifies a process. The ID is assigned when the process is created, and is made available for reassignment when the process terminates. A great many system calls require process IDs as input. Do not confuse this with a *userID*, which is a system for keeping track of memory allocation by various parts of the system, and is handled (pardon the pun) by the Memory Manager tool set. Also, do not confuse Memory Manager *userID*'s with Unix *user ID*'s - numbers which are assigned to the various human users of a multiuser machine.

There are two methods for creating new processes: the system call **fork()** and the library routine **exec()** (specifics for calling these functions and others is in Appendix A **Making System Calls**). **fork** starts up a process which begins execution at an address you specify. **exec** starts up a process by loading an executable file (S16 or EXE). **fork** is used mainly for use inside a specific application, such as running shell built-ins in the background, or setting up independent entities inside a program. Forked processes have some limitations, due to the hardware design of the Apple IIgs. The parent process (the process which called **fork**) must still exist when the children die, either via **kill()** or by simply exiting. This is because the forked children share the same memory space as the parent; the memory the children execute from is tagged with the parent's *userID*. If the parent terminated before the children, the children's code would be deallocated and likely overwritten. A second caveat with **fork** is the difference between it's UNIX counterpart. UNIX **fork** begins executing the child at a point directly after the call to **fork**. This cannot be accomplished on the Apple IIgs because virtual memory is required for such an operation; thus the need to specify a **fork** child as a C function. Note that an appropriately written assembly language program need not necessarily have these restrictions. When a process is forked, the child process is given it's own direct page and stack space under a newly allocated *userID*, so that when the child terminates this memory is automatically freed.

exec is used when the process you wish to start is a GS/OS load file (file type S16 and EXE). **exec** follows the procedure outlined in the *GS/OS Reference Manual* for executing a program, and sets up the new program's environment as it expects. After **exec** has loaded the program and set up its environment, the new process is started and **exec** returns immediately.

Both **fork** and **exec** return the process ID of the child. The parent may use this process ID to send *signals* to the child, or simply wait for the child to exit with the **wait** system call; indeed, this is the most common use. Whenever a child process terminates or is stopped (See Chapter 6 **Signals**), the kernel creates a packet of information which is then made available to the process' parent. If the parent is currently inside a wait call, the call returns with the information. If the parent is off doing something else, the kernel sends the parent process a SIGCHLD signal. The default is to ignore SIGCHLD, but a common technique is to install a handler for SIGCHLD, and to make a **wait** call inside the handler to retrieve the relevant information.

exec is actually implemented as two other system calls: **fork**, and one called **execve**. **execve** loads a program from an executable file, and begins executing it. The current process' memory is deallocated. The shell uses a **fork()/execve()** pair explicitly, so it can set up redirection and handle job control.

Process Table

Information about processes is maintained in the process table, which contains one entry for each possible process (**NPROC**, defined in the C header file `<gno/conf.h>`). There is other per-process information spread about the kernel, but those are usually used for maintaining compatibility with older software, and thus are not described here. Please note that the data in this section is informational only (e.g. for programs like 'ps'). Do not attempt to modify kernel data structures or the GNO Kernel will likely respond with a resounding crash. Only 'interesting' fields are documented.

Copies of process entries should be obtained by using the Kernel Virtual Memory (KVM) routines (`kvm_open`, etc.) These are documented in the electronic manual pages.

processState - processes have a state associate with them. The state of the process is a description of what the process is doing. The possible process states (as listed in `<gno/proc.h>` and described here) are:

RUNNING	the process is currently in execution.
READY	the process is not currently executing, but is ready to be executed as soon as it is assigned a time slice.
BLOCKED	the process is waiting for a slow I/O operation to complete (for instance, a read from a TTY).
NEW	the process has been created, but has not executed yet.
SUSPENDED	the process was stopped with SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU.
WAITING	the process is waiting on a semaphore 'signal' operation. Programs waiting for data from a pipe have this state.
WAITSIGCH	the process is waiting to receive a SIGCHLD signal.
PAUSED	the process is waiting for any signal.

ttyID the device number of the controlling TTY for this process. This is not a GS/OS refnum; rather, it is an index into the kernel's internal character device table. See below for a mapping of ttyIDs to devices.

ticks the number of full ticks this process has executed. If a process gives up its time slice due to an I/O operation, this value is not incremented. A tick is 1/60 second.

alarmCount

if an **alarm(2)** request was made, this is the number of seconds remaining until the process is sent **SIGALRM**.

openFiles this is a structure which stores information about the files a process has open. See **struct ftable** and **struct fdentry** in `<gno/proc.h>`.

irq_A, irq_X, irq_Y, irq_S, irq_D, irq_B, irq_P, irq_state, irq_PC, irq_K

Context information for the process. These fields are the values of the 65816 registers at the last context switch. They only truly represent the machine state of the process if the process is not **RUNNING**.

args a NUL-terminated (C-style) string that contains the command line the process was invoked with. This string begins with 'BYTEWRKS', the shell identifier.

For more details and an example of how to investigate process information, look at the source code for 'GNO Snooper CDA' (**GNOSnooper.c**).

The value in **ttyID** can be interpreted as follows:

- 0 - .null
- 1 - .ttya
- 2 - .ttyb
- 3 - .ttyco
- 6 - .ptyq0 pty0 master side
- 7 - .ttyq0 pty0 slave side

Other values may be appropriate depending on the **tty.config** file. Namely, 1 and 2 (by default the modem and printer port drivers), and 4 and 5 (unassigned by default) may be assigned to different devices.

Task Switching

As mentioned earlier, user code can often unwittingly initiate a context switch by reading from the console (and other miscellaneous things). There are a few situations where this can cause a problem, namely inside interrupt handlers. While the kernel makes an attempt to prevent this, it cannot predict every conceivable problem. The kernel attempts to detect and prevent context switches inside interrupt handlers by checking for the following situations.

- Is the system busy flag non-zero? (the busy flag is located at address `$E100FF`).
- Is the "No-Compact" flag set? (Located at `$E100CB`)
- Does the stack pointer point to `$0100-$01FF`?
- Is the interrupt bit in the processor status register set?

If any of these conditions are met, a context switch will not take place. This can cause problems in certain circumstances. The basic rule is to avoid making Kernel calls that might cause a context switch or change in process state from inside an interrupt handler. This includes the following:

reading from the console or accessing a pipe

wait(), **pause()**, **sigpause()**, **kill()**, **fork()**, **execve()**, **receive()**

Calls such as **send()**, however, are okay to use from an interrupt handler, and in fact are very useful in such situations.

Job Control

Job control is a feature of the kernel that helps processes orderly share a terminal. It prevents such quandaries as "What happens when two processes try to read from the terminal at the same time?".

Job control works by assigning related processes to a *process group*. For example, all of the processes in a pipeline belong to one process group. Terminal device drivers also belong to process groups, and when the process group of a job does not match that of its *controlling terminal* the job is said to be in the background. Background jobs have access to their controlling terminal restricted in certain ways.

- If a background job attempts to read from the terminal, the kernel suspends the process by sending the `SIGTTIN` signal.
- The interrupt signals `SIGTSTP` and `SIGINT`, generated by `^Z` and `^C` respectively, are sent only to the foreground job. This allows backgrounded jobs to proceed without interruption.
- Certain `ioctl()` calls cannot be made by a background job; the result is a `SIGTTIN` signal.

Job control is accessed by software through the `tcnewpgrp`, `tctpgrp`, and `settpgrp` system calls. See the `JOB CONTROL(2)` and `ioctl(2)` manpages.

Chapter 6

Interprocess Communication

"Oh, give me a home where the semaphores roam, and the pipes are not deadlocked all day..."
Unknown western hero

The term Interprocess Communication (*IPC*) covers a large range of operating system features. Any time a process needs to send information to another process some form of IPC is used. The GNO Kernel provides several basic types: semaphores, signals, pipes, messages, ports, and pseudo-terminals. These IPC mechanisms cover almost every conceivable communication task a program could possibly need to do.

Semaphores

In the days before radio, when two ships wished to communicate with each other to decide who was going first to traverse a channel wide enough only for one, they used multicolored flags called semaphores. Computer scientists, being great lovers of anachronistic terms, adopted the term and meaning of the word semaphore to create a way for processes to communicate when accessing shared information.

GNO/ME, like other multitasking systems, provides applications with semaphore routines. Semaphores sequentialize access to data by concurrently executing processes. You should use semaphores whenever two or more processes want to access shared information. For example, suppose there were three processes, each of which accepted input from user terminals and stored this input into a buffer in memory. Suppose also that there is another process which reads the information out of the buffer and stores it on disk. If one of the processes putting information in the buffer (writer process) was in the middle of storing information in the buffer when a context switch occurred, and one of the other processes then accessed the buffer, things would get really confused. Code that accesses the buffer should not be interrupted by another process that manipulates the buffer; this code is called a *critical section*; in order to operate properly, this code must not be interrupted by any other attempts to access the buffer.

To prevent the buffer from becoming corrupted, a semaphore would be employed. As part of it's startup, the application that started up the other processes would also create a semaphore using the `screate(2)` system call with a parameter of 1. This number means (among other things) that only one process at a time can enter the critical section, and is called the *count*.

When a process wishes to access the buffer, it makes a `swait(2)`, giving as argument the semaphore number returned by `screate`. When it's done with the buffer, it makes an `ssignal(2)` call to indicate this fact.

This is what happens when `swait` is called: the kernel first decrements the count. If the count is then less than zero, the kernel suspends the process, because a count of less than zero indicates that another process is already inside a critical section. This suspended state is called 'waiting' (hence the name of `swait`). Every process that tries to call `swait` with count < 0 will be suspended; a queue of all the processes currently waiting on the semaphore is associated with the semaphore.

Now, when the process inside the critical section leaves and executes `ssignal`, the kernel increments the count. If there are processes waiting for the semaphore, the kernel chooses one arbitrarily and restarts it. When the process resumes execution at its next time slice, its `swait` call will finish executing and it will have exclusive control of the critical section. This cycle continues until there are no processes waiting on the semaphore, at which point its count will have returned to 1.

When the semaphore is no longer needed, you should dispose of it with the **sdelete(2)** call. This call frees any processes that might be waiting on the semaphore and returns the semaphore to the semaphore pool.

One must be careful in use of semaphores or *deadlock* can occur.

There are (believe it or not) many situations in everyday programming when you may need semaphores, moreso than real UNIX systems due to the Apple IIgs's lack of virtual memory. The most common of these is your C or Pascal compiler's stdio library; these are routines like **printf** and **writeln**. In many cases, these libraries use global variables and buffers. If you write a program which forks a child process that shares program code with the parent process (i.e. doesn't **execve** to another executable), and that child and the parent both use non-*reentrant* library calls, the library will become confused. In the case of text output routines, this usually results in garbaged output.

Other library routines can have more disastrous results. For example, if a parent's **free()** or **dispose()** memory management call is interrupted, and the child makes a similar call during this time, the linked lists that the library maintains to keep track of allocated memory could become corrupted, resulting most likely in a program crash.

GNO/ME provides *mutual exclusion* (i.e., lets a maximum of one process at a time execute the code) automatically around all Toolbox and GS/OS calls as described in Chapter 3, and also uses semaphores internally in many other places. Any budding GNO/ME programmer is well advised to experiment with semaphores to get a feel for when and where they should be used. Examples of semaphore use can be found in the sample source code, notably `dp.c` (Dining Philosophers demo) and `pipe*.c` (a sample implementation of pipes written entirely in C).

Signals

Another method of IPC is software signals. Signals are similar to hardware interrupts in that they are asynchronous; that is, a process receiving a signal does not have to be in a special mode, does not have to wait for it. Also like hardware interrupts, a process can install signal handlers to take special action when a signal arrives. Unlike hardware interrupts, signals are defined and handled entirely through software.

Signals are generally used to tell a process of some event that has occurred. Between the system-defined and user-defined signals, there is a lot of things you can do. GNO/ME currently defines 32 different signals. A list of signals and their codes can be found in **signal(2)** and the header file `<gno/signal.h>`.

There are three types of default actions that occur upon receipt of a signal. The process receiving the signal might be terminated, or stopped; or, the signal might be ignored. The default action of any signal can be changed by a process, with some exceptions. Not all of the defined signals are currently used by GNO/ME, as some are not applicable to the Apple IIgs, or represent UNIX features not yet implemented in GNO/ME. Here is a list of the signals that are used by GNO/ME.

SIGINT	This signal is sent to the foreground job when a user types ^C at the terminal keyboard.
SIGKILL	The default action of this signal (termination) cannot be changed. This provides a sure-fire means of stopping an otherwise unstoppable process.
SIGPIPE	Whenever a process tries to write on a pipe with no readers, it is sent this signal.
SIGALRM	SIGALRM is sent when an alarm timer expires (counts down to zero). An application can start an alarm timer with the alarm(2) system call.

SIGTERM	This is the default signal sent by kill(1) . Use of this signal allows applications to clean up (delete temporary files, free system resources like semaphores, etc) before terminating at the user's bequest.
SIGSTOP	This signal is used to stop a process' execution temporarily. Like SIGKILL, processes are not allowed to install a handler for this signal.
SIGCONT	To restart a stopped process, send this signal.
SIGTSTP	This is similar to SIGSTOP, but is sent when the user types ^Z at the keyboard. Unlike SIGSTOP, this signal can be ignored, caught, or blocked.
SIGCHLD	A process receives this signal whenever a child process is stopped or terminates. gsh uses this to keep track of jobs, and the wait system call waits for this signal to arrive before exiting.
SIGTTIN	This signal also stops a process. It is sent to background jobs that try to get input from the terminal.
SIGTTOU	Similar to SIGTTIN, but is sent when a background process tries to write to the terminal. This behavior is optional and is by default turned off.
SIGUSR1 SIGUSR2	These two signals are reserved for application authors. Their meaning will change from application to application.

As you can see, signals are used by many aspects of the system. For detailed information on what various signals mean, consult the appropriate electronic manual page - see **tty(4)**, **wait(2)**, and **signal(2)**.

For an example of signal usage, consider a print spooler. A print spooler takes files that are put in the spool directory on a disk and sends the data in the files to a printer. There are generally two parts to a print spooler: the *daemon*, a process that resides in memory and performs the transfer of data to the printer in the background; and the spooler. There can be many different types of spoolers, say one for desktop printing, one for printing source code, etc. To communicate to the daemon that they have just placed a new file in the spool directory, the spoolers could send the daemon SIGUSR. The daemon will have a handler for SIGUSR, and that handler will locate the file and set things up so the print will begin. Note that the actual implementation of the print spooling system in GNO/ME, **lpr(1)** and **lpd(8)**, is somewhat more complex and uses messages and ports instead of signals. However, an earlier version of the spooler software *did* use signals for communication.

Signals should not be sent from inside an interrupt handler, nor from inside a GS/OS or Toolbox call. Window Manager update routines are a prime example of code that should not send signals; they are executed as part of a tool call. The GS/OS aspect of this limitation is a little harder to come up against. GS/OS does maintain a software signal facility of it's own, used to notify programs when certain low-level events have occurred. Do not confuse these GS/OS signals with GNO/ME signals, and above all, don't send a GNO/ME signal from a GS/OS signal handler.

When a process receives a signal for which it has installed a handler, what occurs is similar to a context switch. The process' context is saved on the stack, and the context is set so that the signal handler routine will be executed. Since the old context is stored on the stack, the signal handler may if it wishes return to some other part of the program. It accomplishes this by setting the stack pointer to a value saved earlier in the program and jumping to the appropriate place. Jumps like this can be made with C's **setjmp** and **longjmp** functions. The following bit of code demonstrates this ability.

```
void sighandler(int sig, int code)
{
    printf("Got a signal!");
    longjmp(jmp_buf);
}
```



```

void routine(void)
{
    signal(SIGUSR, sighandler);
    if (setjmp(jmp_buf)) {
        printf("Finally done! Sorry for all that....");
    }
    else { while (1) {
        printf("While I wait I will annoy you!");
    }
}
}

```

This program basically prints an annoying message over and over until SIGUSR is received. At that point, the handler prints "Got a Signal!" and jumps back to the part of the if statement that prints an apology. If the signal handler hadn't made the **longjmp**, when the handler exited control would have returned to the exact place in the **while(1)** loop that was interrupted.

Similar techniques can be applied in assembly language.

Pipes

This third form of IPC implemented in GNO/ME is one of the most powerful features ever put into an operating system. A pipe is a conduit for information from one process to another. Pipes are accessed just like regular files; the same GS/OS and ToolBox calls currently used to manipulate files are also used to manipulate pipes. When combined with GNO/ME standard I/O features, pipes become very powerful indeed. For examples on how to use **gsh** to connect applications with pipes, see the *GNO Shell Reference Manual*.

Pipes are uni-directional channels between processes. Pipes are created with the **pipe(2)** system call, which returns two GS/OS refNums; one for the write end, and one for the read end. An attempt to read from the write end or vice-versa results in an error.

Pipes under GNO/ME are implemented as a circular buffer of 4096 bytes. Semaphores are employed to prevent the buffer from overflowing, and to maintain synchronization between the processes accessing the pipe. This is done by creating two semaphores; their counts indicate how many bytes are available to be read and how many bytes may be written to the buffer (0 and 4096 initially). If an I/O operation on the pipe would result in the buffer being emptied or filled, the calling process is blocked until the data (or space) becomes available.

The usual method of setting up a pipeline between processes, used by **gsh** and utilities such as **script**, is to make the **pipe()** call and then **fork** off the processes to be connected by the pipe.

```

int fd[2];
int testPipe(void)
{
    pipe(fd);           /* create the pipe */
    fork(proc1);        /* create the writer process */
    fork(proc2);        /* create the reader process */
    close(fd[1]);        /* we don't need the pipe anymore, because */
    close(fd[0]);        /* the children inherited them */
    { wait for processes to terminate ... }
}

```



```

void proc1(void)
{
    dup2(STDOUT_FILENO, fd[1]);    /* reset standard output to the write pipe */
    close(fd[0]);                 /* we don't need the read end */
    { exec writer process ... }
}
void proc2(void)
{
    dup2(STDIN_FILENO, fd[0]);    /* reset standard input to the pipe */
    close(fd[1]);                 /* we don't need the write end */
    { exec reader process ... }
}

```

Recall that when a new process is forked, it inherits all of the open files of its parent; thus, the two children here inherit not only standard I/O but also the pipe. After the forks, the parent process closes the pipe and each of the child processes closes the end of the pipe it doesn't use. This is actually a necessary step because the kernel must know when the reader has terminated in order to also stop the writer (by sending `SIGPIPE`). Since each open refNum to the read end of the pipe is counted as a reader, any unnecessary copies must be closed.

For further examples of implementing and programming pipes, see the sample source code for `pipe*.c`.

Messages

GNO's Message IPC is borrowed from the XINU Operating System, designed by Douglas Comer. It is a simple way to send a datum (a message) to another process. Messages are 32-bit (4-byte) longwords.

The Message IPC is centered around two calls, **send()** and **receive()**. **send()** sends a message to a specified process ID. To access that message, a process must use **receive()**. If no message is waiting for a process when it calls **receive()**, it is put to sleep until a message becomes available.

Since a process can only have one pending message, the Message IPC is useful mostly in applications where two or more cooperating processes only occasionally need to signal each other; for example, the **init(8)** program communicates with the Init daemon by sending messages. Various attributes are encoded in the 32-bit value sent to **init(8)** to instruct it on how to change its state.

If a process doesn't want to indefinitely block waiting for a message, it can call **recvtim()**. **recvtim()** accepts a timeout parameter which indicates the maximum amount of time to wait for a message.

Ports

GNO/ME Ports IPC can be thought of as an extended version of Messages. Whereas only one message can be pending at once, a port can contain any number of pending messages (up to a limit defined when an application creates a port).

Like Messages, Ports transmit 32-bit values between processes. The calls **psend()** and **preceive()** work similarly to their Message counterparts.

A Port is created with the **pcreate()** call. The application specifies the size of the port in this call. When the application is done with the port, it should call **pdelete()** to free up the resources used by the port.

One of the most important aspects of ports is the ability to bind a *name* to a port. Whereas many of GNO/ME IPC mechanisms require the communicating processes to be related in some way (common children of the same parent, for instance) being able to give a port a name means that totally unrelated processes can communicate. For example, the GNO/ME print spooling system uses a named port for communicating information about the addition of new jobs to the print queue. The printer daemon, **lpd**(8), creates a port with a specific name; the name is defined by the author of the print daemon; any application that wishes to have the daemon print a spool file also knows this name. (The standard print daemon uses the name "LPDPrinter"). The name allows an application to find lpd's port regardless of the actual numeric port ID (which might be different from system to system, or even from session to session on the same machine).

Names are bound to ports with the **pbind()** call. The numeric port ID can be obtained by passing a name to **pgetport()**.

Pseudo-Terminals (PTYs)

Pseudo-terminals are a bi-directional communication channel that can be used to connect two processes (or more correctly, a process group to another process). You may (correctly) ask why two pipes would not do the same thing; the answer is that a lot of modern UNIX software relies on the way the terminal interface works, and thus would malfunction when presented with a pipe as standard input. What PTYs provide is a lot like two pipes, but with a TTY interface.

PTYs can be used in a number of important and exciting applications, such as windowing systems and 'script-driven' interfaces.

Windowing systems like the UNIX X-Windows use PTYs to give a process group an interface that looks exactly like a real terminal; however, the 'terminal' in this case is actually a window in a graphics-based system. The program that manages the window ('xterm' in X-Windows) is called the *master*. It is responsible for setting up the PTY, and starting up the process with redirection (usually a shell) that is to run in the window. The process running in the window is called the *slave*.

To allocate a PTY, the master opens in turn each PTY device starting with `.ptyq0`. If a PTY is already in use, the open call will return an error (the kernel uses the EXCL flag internally). When an open succeeds, the master then has exclusive access to that PTY. At this point, the master opens the corresponding TTY file (`.ttyq0` - `.ttyqf`), or the slave device. It then forks off a process, which sets redirection up in the normal fashion and then exec's the program to run on the PTY. The following is taken from the source code for the Graphical Shell Interface (GSI) NDA.

`initPipe` scans the PTY devices, looking for a free one as discussed above. Note that the master side of a PTY does *not* have (by default) a terminal interface; it is a very raw device, with only a few `ioctl`'s to be able to send signals and handle other such low-level tasks.

```
char buffer[1024];
int ptyno, master;

int initPipe(void)
{
    int cl[2];
    struct sgttyb sb;
    char *ptyname = ".ptyq0";
    unsigned i;

    /* We have to open the master first */

    for (i = 0; i < 2; i++) {
```



```

        ptyname[5] = intToHex(i); /* generate a PTY name from the index */
        master = open(ptyname,O_RDWR);
        if (master > 0) break; /* successful open */
    }
    if (master < 1) { return -1; }
    ptyno = i;
    pid1 = fork(producer);
    return 0;
}

```

`producer()` sets up redirection for the shell, and also opens the slave side of the PTY. The slave processes must not have any access whatsoever to the master side of the PTY, so `close(0)` is used to close all open files (which includes, at this point, the master PTY file descriptor from `initPipe`). Note that as in many pipe applications, the file descriptor that will be assigned to a newly opened file is assumed, and that can be safely done in this case because it is clear that with no files open the next file descriptor will be 1.

```

/* the shell is executed here */
#pragma databank 1
void producer(void)
{
    char *ptyname = ".ttyq0";

    /* we must not have access to ANY other ttys */
    close(0); /* close ALL open files */
    ptyname[5] = intToHex(ptyno);
    /* modify the tty slave name to correspond to the master */
    slave = open(ptyname,O_RDWR); /* file descriptor 1 */
    dup(slave); /* fd 2 */
    dup(slave); /* fd 3 */
    SetOutputDevice(3,21); /* Set up the TextTools redirection */
    SetErrorDevice(3,31);
    SetInputDevice(3,11);
    WriteCString("Welcome to GNO GSI\r\n");
    execve(":bin:gsh","gsh -f");
    /* If we get here, we were unable to run the shell */
    printf("Could not locate :bin:gsh : %d",errno);
}
#pragma databank 0

```

`consume()` is called as part of GSI's event loop. It simply checks to see if there is any data for the master by using the `FIONREAD` ioctl, one of the few ioctl's supported by the master side. See `PTY(4)` for details. Any data that is available is sent to the window via a routine `toOut`, which inserts the new data into a `TextEdit` record.

```

void consume(CtlRecHndl teH)
{
    char ch;
    int fio,fio1,i;

    ioctl(master,FIONREAD,&fio);
    if (fio) {
        if (fio > 256) fio = 256;
        fio1 = read(master,buffer,fio);
        buffer[fio] = 0;
        toOut(buffer,fio,teH);
        updateWind1(fio,fio1);
    }
}

```



```
}
```

When the user types a key, the keypress is sent to the slave by simply writing the data with a write call.

```
void writedata(char k)
{
    write(master, &k, 1);
}
```

When the user is done with the window and closes it, GSI closes the master end of the PTY.

```
void closePipe(void)
{
    int cl[2];

    close(master);
}
```

When this is done, the slave process receives a SIGHUP signal, to indicate that the connection was lost. Since the standard behavior of SIGHUP is to terminate the process, the slave dies and either the slave or the kernel closes the slave end. At this point, the PTY is available for re-use by another application.

As you can see, PTYs are very simple to program and use. The simplicity can be misleading, for PTYs are a very powerful method of IPC. As another example of the use of PTYs, we point out that PTYs can be used to drive programs with 'scripts'. These scripts are a series of 'wait-for' and 'print' operations, much like auto-logon macros in communications programs such as ProTERM. Script-driving a program can be used to automate testing or use of an application.

PTYs can be used to test software that would normally work over a regular terminal (such as a modem). Since PTYs are identical (to the slave) to terminals, the application being tested doesn't know the difference. What this means to the programmer is incredible power and flexibility in testing the application. For example, a communications program could be nearly completely tested without ever dialing to another computer with a modem!

There are so many applications of PTYs that to attempt to discuss them all here would be impossible; as PTYs are discovered by more GNO/ME programmers we expect that more useful PTY applications will become available.

Deadlock

With interprocess communication comes the problem of *deadlock*. If a situation arises where two or more processes are all waiting for an signal from one of the other waiting processes, the processes are said to be deadlocked.

The best way to explain deadlock is to give an example. Suppose that two processes are connected with two pipes so that they can communicate bidirectionally. Also suppose that each of the pipes are full, and that when each process writes into one of the pipes they are blocked. Both processes are blocked waiting for the other to unblock them.

There is no way for the operating system to detect every conceivable deadlock condition without expending large amounts of CPU time. Thus, the only way to recover from a deadlock is to kill the processes in question. Responsibility for preventing deadlock situations is placed on the programmer. Fortunately, situations where deadlock can occur are infrequent; however, you should keep an eye out for them and try to work around them when they do occur.

Appendix A

Making System Calls

The GNO Kernel is accessed through system calls. The actual procedure is very simple from C: simply #include the appropriate header file as noted in the synopsis of the call's manual page, and call it as you would any other C function. From assembly language the procedure is no more difficult, using the advanced macros provided for the APW and ORCA assemblers. Make sure, however, that you have defined a word variable **errno**. Lowercase is important, use the 'case on' and 'case off' directives to ensure that the definition of **errno** is case-sensitive. The system call interface libraries store any error codes returned by the kernel in this variable.

If you are going to be accessing the kernel from a language other than those for which interfaces are provided, then the following information is for you.

System Call Interface

The system calls are implemented as a user toolset, tool number 3. These tools are called the same way regular system tools (such as QuickDraw) are called, except that you must JSL \$E10008 instead of JSL \$E10000 (or \$E1000C instead of \$E10004 for the alternate entry point). The function numbers for the currently defined tools are as follows:

getpid	\$0903	kill	\$0A03
fork	\$0B03	swait	\$0D03
ssignal	\$0E03	screate	\$0F03
sdelete	\$1003	kvm_open	\$1103
kvm_close	\$1203	kvm_getproc	\$1303
kvm_nextproc	\$1403	kvm_setproc	\$1503
signal	\$1603	wait	\$1703
tcnewpgrp	\$1803	setpgrp	\$1903
tctpgrp	\$1A03	sigsetmask	\$1B03
sigblock	\$1C03	execve	\$1D03
alarm	\$1E03	setdebug	\$1F03
setsystemvector	\$2003	sigpause	\$2103
dup	\$2203	dup2	\$2303
pipe	\$2403	getpgrp	\$2503
ioctl	\$2603	stat	\$2703
fstat	\$2803	lstat	\$2903
getuid	\$2A03	getgid	\$2B03
geteuid	\$2C03	getegid	\$2D03
setuid	\$2E03	setgid	\$2F03
send	\$3003	receive	\$3103
recvclr	\$3203	recvtim	\$3303
setpgrp	\$3403	times	\$3503
pcreate	\$3603	psend	\$3703
preceive	\$3803	pdelete	\$3903
preset	\$3A03	pbind	\$3B03
pgetport	\$3C03	pgetcount	\$3D03
scount	\$3E03	fork2	\$3F03

getppid
alarm10

\$4003
\$4203

SetGNOQuitRec

\$4103

Parameters should be pushed onto the stack in the same order as defined by the C prototypes outlines in the synopsis section of the manual pages; that is, left-to-right. In addition to those parameters, all of the functions (except those denoted by a *) take an integer pointer parameter **ERRNO**. This is a pointer to a word value which will contain the **errno** code returned by the function if an error occurs, and should be pushed onto the stack after all the other parameters. The calls do not clear this code to 0 if no error occurs; thus, you must check the return value of the function to see if an error occurred, and then check **errno** to get the actual error code.

Do not forget to also push space on the stack (before the parameters) for the call to store its return value.

These low-level system call interfaces are not to be used in general programming. It is assumed the programmer will use the libraries provided, or use this information to create a new library. The system call interface is subject to change without notice; any changes will, of course, be documented in future versions of GNO/ME.

System Call Error Codes

The following codes are taken from `<errno.h>`. The codes up to **EPERM** are the same values as those defined by ORCA/C for compatibility reasons. Error conditions are usually reported by system calls by returning a -1 (word) or NULL (long) value. Which error codes can be expected from a particular call are detailed in the errors section in the appropriate manual page.

EDOM	domain error. Basically an undefined error code.
ERANGE	Range error. A value passed to a system call was too large, too small, or illegal.
ENOMEM	Not enough memory. The kernel could not allocate enough memory to complete the requested operation.
ENOENT	No such file or directory. The file specified could not be found.
EIO	I/O error. An error occurred trying to perform an I/O operation (could be bad media). Also refers to a disk error not covered by the other errno codes.
EINVAL	Invalid argument. An argument to a system call was invalid in some way.
EBADF	bad file descriptor. The file descriptor passed to the kernel does not represent an open file.
EMFILE	too many files are open. The kernel cannot open any more files for this process; it's open file table is full. Close some other open files to retry the operation.
EACCESS	access bits prevent the operation. One of the access bit settings (delete, rename, read, write) associated with the file does not allow the requested operation.
EEXIST	the file exists. An attempt to create a new file with the same name as an existing file results in this error.
ENOSPC	No space on device. There is not enough room on the requested device to complete the operation. This is usually indicative of a full disk.
EPERM	Not owner. Not yet used in GNO.
ESRCH	No such process. The process ID specified does not refer to an active process. Possibly the process terminated earlier.
EINTR	Interrupted system call. Certain system calls can be interrupted by signals. In cases where the user has specified that those calls not be automatically restarted, the call will return this error.
E2BIG	Arg list too long. Too many arguments were specified in an <code>execve</code> calls.
ENOEXEC	Exec format error. The file specified is not in an executable format (OMF load file).

ECHILD	No children. This error is returned by wait(2) when there are no child processes left running.
EAGAIN	No more processes. The process table is full, the fork(2) cannot complete.
ENOTDIR	Not a directory. One of the elements in a pathname refers to a file which is not a directory.
ENOTTY	Not a terminal. The file descriptor passed to an ioctl(2) or job control call does not refer to a terminal file.
EPIPE	Broken pipe. If a process attempts to write on a pipe with no readers, and has blocked or ignored SIGPIPE, this error is returned by the write operation.
ESPIPE	Illegal seek. Similar to ENOTBLK, but specific for pipes.
ENOTBLK	not a block device. An attempt to perform an operation on a character device that only makes sense on a block device, e.g. creating a file.

System Panics

In most cases, if the kernel detects an error in operation an appropriate error code is returned by the function in question (GS/OS calls, ToolBox calls, or system calls as described above). However, there are rare circumstances where the kernel detects what should be an impossible condition. This can happen due to bugs in the kernel, because the kernel was overwritten by a buggy program, or for any number of other reasons.

When the kernel does come across such an error, system operation cannot continue and what ensues is called a *system panic*. Panics are very easily noticed- the kernel will print an error message on the screen and ensure that the text screen is visible, turning off any graphics mode if necessary. The kernel then sets the text and background colors to red on white - a very noticeable condition. At that point, the kernel turns off context switching to prevent any background process or other interrupt driven code from further confusing the system. This is done mainly to prevent damage to disk directory structures by a bad system.

When a system panic does occur, the only thing you can do is reboot your system. If you can reliably reproduce a system panic, please record the panic message and the sequence of events necessary to evoke the panic and report the information to Procyon, Inc.

Appendix B

Miscellaneous Programming Issues

Option Arguments

The Free Software Foundation (known as the FSF), invented user friendly long format option arguments, and defined the "+" character for interpretation that a long format follows. This interpretation is generally followed in the UNIX community. There are two files which will assist you in programming GNO/ME utilities with both short and long format options, "getopt.h" for short options, and "getopt1.h" for long options.

Pathname Expansion

Those of you familiar with programming in the ORCA environment should be familiar with the shell calls `InitWildcard` and `NextWildcard`. These shell calls, while supported by **gsh**, are no longer necessary. All shell utilities that work with multiple filenames do not need to provide support for file globbing, as this is taken care of transparently to the command.

Glossary

The following terms usually have references in the main text, indicated by italics.

<i>Asynchronous</i>	An event that may take place at any time. See synchronous.
<i>BASIC.</i>	Beginners All-purpose Symbolic Instruction Code. A simple computer language.
<i>Blocked</i>	Refers to a process waiting for some event to occur. Processes can block on terminal I/O, signals, and other IPC and I/O functions.
<i>Console</i>	The terminal which represents the IIGS's keyboard and monitor.
<i>Context</i>	The attributes which define the state of a process. This includes the program counter, stack pointer, and other machine registers (both CPU and other computer hardware).
<i>Controlling terminal</i>	The terminal which 'controls' a process or process group; processes can receive keyboard signals (such as SIGTSTP, or ^Z) only from their controlling terminal.
<i>Critical section</i>	A piece of code inside which only one process at a time may be allowed to execute. Critical sections are usually protected by semaphores.
<i>Daemon</i>	A process that runs in the background and waits to act on an asynchronous event. These can be anything: waiting for a caller on a modem, waiting for spooled files to print, etc. Daemons are usually started at boot time by the <i>init</i> (8) process.
<i>Deadlock</i>	A situation where two or more communicating processes are blocked, waiting on each other. See Chapter 5, "Deadlock".
<i>Errno</i>	A variable which holds a descriptive numeric error code, returned from C libraries and system calls.
<i>Foofoo {foo, bar}</i>	foofoo derives from an old military acronym FUBAR. In it's politest interpretation it stands for Fouled Up Beyond All Recognition. Computer scientists borrowed the term and created foofoo. When a name for an object is needed but the name itself is not important, foo and bar are first choice among computing science types.
<i>Executable</i>	A program as it resides on disk. Executables can be compiled or assembled programs, or shell scripts. Executables are run by typing their name on the shell's command line and frequently take paramters to determine what data they operate on and particulars of how they do it.
<i>GNO/ME.</i>	GNO Multitasking Environment. The complete package including the GNO kernel and the GNO Shell.
<i>GNO Kernel.</i>	Heart of GNO/ME. Executes processes when asked by the GNO Shell
<i>GNO Shell.</i>	Provides an interface between the user and the GNO kernel.
<i>gsh.</i>	GNO Implementation of a UNIX-like shell.
<i>GS/OS.</i>	16 bit Operating System for the Apple IIGs.
<i>IPC</i>	"Inter-Process Communication". Any method by which processes can pass information to other processes.
<i>Job</i>	A set of related processes. Jobs are generally composed of processes with a common parent and the same controlling terminal.
<i>Manpage</i>	Refers to the system call and utility documentation provided with GNO. Manpages exist on disk in pre-formatted source form (AppleWorks GS currently), and can be viewed by various utilites on a variety of output devices.
<i>Master</i>	Refers to the .PTYxx side of a pseudo-terminal, and also the process controlling that device. The master is usually responsible for setting up the PTY and running a process on it.
<i>Message</i>	A 32-bit value that is passed via the Messages IPC mechanism to another process.

Mutex	Short for mutual exclusion, a term that refers to protecting a critical section.
Panic	An unrecoverable kernel error, usually indicating that an internal data structure has become corrupted.
Parent	When talking about a process, the parent of a process is the one that spawned it; i.e., made the fork() call.
Pipe	A unidirectional IPC mechanism. Pipes transmit binary 8-bit data.
Pipeline	Two or more processes connected by pipes.
Port	A flow-controlled IPC mechanism that can pass longwords of data.
Process	A program in execution.
Process group	An identifying code for a job. Process groups are also assigned to TTY's, which allows the TTY to differentiate background jobs from foreground jobs when sending interrupt signals.
Pseudo-terminal	A bidirectional communications channel, normally used in windowing systems or for advanced control and testing applications.
PTY	See 'pseudo-terminal'.
Semaphore	A data object used to synchronize concurrent processes.
Sequentialization	The task of ensuring that critical sections are only executed by one concurrent process at a time.
Signal	A software interrupt and IPC mechanism.
Slave	1. A good term to describe the relationship of Joe Citizen to the IRS. 2. The .TTYxx side of a pseudo-terminal; the slave is usually an application program of some kind, like a shell.
Suspended	Refers to a process whose execution has been stopped.
Synchronous	An event that takes place at a predetermined time or sequence of times. Also used to indicate the act of waiting for an event to happen. See asynchronous.
Terminal	Any device that looks like a terminal; this includes pseudo-ttys. By definition, a terminal supports all of the tty(4) ioctl calls.
Tty	Short for Teletype. TTY is an anachronistic term; in modern usage it is taken to mean 'terminal'.
UNIX.	Popular operating system which has growing use in education and business. One of the first operating systems to support multitasking.

Index

- ^C 20, 22
- + 32
- . 9
- ^Z 20
- ^Z 23
- .. 9
- .CONSOLE 5, 6, 13
- .d2 10
- .d5 10
- .NULL 11, 19
- .ptyq0 19, 26
- .PTYQ[0-9,A-F] 11
- .ttya 12, 19
- .TTYA[0-9,A-F] 11
- .ttyb 19
- .TTYCO 11, 12, 16, 19
- .ttyq0 19, 26
- .ttyqf 26
- /bin 10
- /etc 10
- /etc/namespace 10
- /etc/tty.config 12
- /etc/ttys 13
- /usr 10
- 65816 processor 6
- 65816 registers 19
- _toolErr 5
- alarm 19, 22, 29
- alarm10 30
- alarmCount 19
- ANDmask 15
- APW 29
- args 19
- Assembly 6
- auxType 7
- background 3, 20, 23
- Bank 0 6
- BASIC 13
- blocked 17, 18
- BYTEWRKS 19
- C compiler 6
- CDA 16
- ChangePath 10
- character device 11, 18
- character devices 11
- Character drivers 15
- character strings 6
- child process 17, 22, 25
- chtyp 7
- ClearBackupBit 10
- Close 10, 11, 27
- conf.h 18
- console 5, 16
- console driver 5, 11, 13, 15, 16
- consume 27
- context 17
- context switch 17, 19, 21, 23, 31
- Control Panel 16
- controlling terminal 16, 20
- count 21
- Create 10
- creative programming 6
- critical section 21
- CTRL-D 15
- CurResourceApp 15
- custom drivers 13
- daemon 23
- deadlock 22, 28
- DELETE 15
- desktop 5, 6, 7, 16
- Destroy 10
- device drivers 6, 20
- DInfo 11
- direct page 6, 9, 17
- disk access 6
- Disk I/O 6
- dispose 22
- DMA 6
- dp.c 22
- driver number 13
- DStatus 11
- dup 15, 29
- dup2 29
- E2BIG 30
- EACCESS 30
- EAGAIN 31
- EBADF 30
- ECHILD 31
- EDOM 30
- EEXIST 30
- EINTR 30
- EINVAL 30
- EIO 30
- EMFILE 30
- ENOENT 30
- ENOEXEC 30
- ENOMEM 30
- ENOSPC 30
- ENOTBLK 31
- ENOTDIR 31
- ENOTTY 31
- EOL 15
- EPERM 30

- EPIPE 31
- ERANGE 30
- errno 29, 30
- errno.h 30
- ESPIPE 31
- ESRCH 30
- Event Manager 7, 16
- EXCL 26
- EXE 7, 17, 18
- exec 17, 18, 26
- execve 11, 12, 18, 19, 22, 29
- exit 12
- ExpandPath 9, 10
- file 11
- file descriptor 11, 13, 27
- File Level 10
- file sharing 7
- FIONREAD 27
- FlexBeep 14
- Flush 10, 12
- fopen 7
- foreground 3
- fork 17, 18, 19, 24, 29, 31
- fork2 29
- free 22
- fstat 29
- GetDirEntry 12
- getegid 29
- GetEOF 12
- geteuid 29
- GetFileInfo 10
- getgid 29
- GetLevel 10
- GetMark 12
- GetName 12
- GetOpenFileRefnum 15
- getopt.h 32
- getopt1.h 32
- getpgp 29
- getpid 29
- getppid 30
- GetPrefix 10
- getuid 29
- GNO compliant 6
- GNO kernel 3, 5, 10, 12, 17, 18, 21, 29
- GNO Snooper CDA 19
- gno.h 5
- GNO/ME 3, 5, 6, 7, 9, 10, 11, 12, 13, 15, 17, 21, 22, 23, 24
- GNO/ME compliance 5
- GNO/ME compliant 5, 6
- GNO/ME drivers 11
- GNO/ME signal 23
- GNOsnooper.c 19
- GS/OS 3, 5, 6, 7, 9, 10, 11, 12, 17, 23, 24
- GS/OS call 9, 10, 11, 12, 22, 31
- GS/OS device 11
- GS/OS drivers 11
- GS/OS errors 15
- GS/OS refNum 24
- GS/OS signal handler 23
- GS/OS signals 23
- gsh 23, 24, 32
- GSI 28
- hardware interrupt 22
- I/O 5, 6, 7, 11, 13, 18, 24, 25
- I/O masks 15
- init 25
- initPipe 26, 27
- InitWildcard 32
- Interprocess Communication 21
- interrupt 31
- interrupt handler 16, 19, 23
- ioctl 6, 11, 20, 27, 29, 31
- IPC 21, 22, 24, 25, 26, 28
- irq_A 19
- irq_B 19
- irq_D 19
- irq_K 19
- irq_P 19
- irq_PC 19
- irq_S 19
- irq_state 19
- irq_X 19
- irq_Y 19
- Job control 20
- kernel 5, 21, 28, 31
- kernStatus 5
- kernVersion 5
- keyboard input 6
- keyboard latch 6
- kill 17, 19, 23, 28, 29
- KVM 18
- kvm_close 29
- kvm_getproc 29
- kvm_nextproc 29
- kvm_open 18, 29
- kvm_setproc 29
- LEFT-ARROW 15
- local arrays 6
- long options 32
- longjmp 23, 24
- lpd 23, 26
- lpr 23
- lstat 29
- main memory 6

- malloc 6
- master 26, 27
- master. 26
- Memory Manager 6, 17
- messages 21, 25
- MiscTool 15
- modem 5, 12, 19
- multitasking 3, 9, 10, 13, 16, 21
- mutual exclusion 22
- NEW 18
- NextWildcard 32
- NPROC 18
- Null device driver 13
- OA-D 15
- OA-E 15
- Open 10, 11
- openFiles 19
- OpenGS 7
- option arguments 32
- ORCA 10, 12, 13, 29, 32
- ORCA/C 6, 30
- ORCA/C 1.2 7
- ORCA/C 1.3 6, 7, 12
- ORCA/C 2.0 6, 12
- ORmask 15
- OSShutdown 12
- panic 15, 31
- parent process 17, 22, 25
- Pascal compiler 22
- Pascal control codes 11
- Pascal driver 13
- Pascal firmware 13
- pause 19
- PAUSED 18
- pbind 26, 29
- pcreate 25, 29
- pdelete 25, 29
- pgetcount 29
- pgetport 26, 29
- pipe 11, 13, 18, 21, 22, 24, 29
- pipe*.c 22, 25
- pipeline 20, 24
- port drivers 12, 19
- port ID 26
- ports 21, 25
- preceive 25, 29
- preemptive multitasking 17
- prefixes 9
- preset 29
- print spooler 23
- printer 12, 19, 23
- printf 22
- proc.h 18, 19
- process 17, 20, 21, 22, 28
- process group. 20
- process ID 17, 25
- process table 18
- processState 18
- ProDOS-16 9
- producer 27
- ps 18
- psend 25, 29
- Pseudo-terminals 12, 21, 26
- PTY 11, 26, 27, 28
- QDStartup 16
- QuickDraw 9, 29
- QUIT 11, 12
- QuitGS 11, 12
- Read 11
- ReadChar 15
- ReadLine 14
- READY 18
- receive 19, 25, 29
- recursion 6
- recvclr 29
- recvtim 25, 29
- redirection 13, 26
- reentrant 22
- refNum 10, 11, 12, 13, 18, 25
- RELEASE.NOTES 7
- requestAccess 7
- resource 15
- Resource Manager 15
- RestAll 16
- restartability 12
- rexit 12
- RIGHT-ARROW 15
- round-robin scheduling 17
- RUNNING 18, 19
- S16 7, 17, 18
- SANE 9
- SaveAll 16
- scount 29
- screate 21, 29
- screen memory 6
- script 24, 28
- sdelete 22, 29
- semaphore 21, 22, 24
- semaphore pool 22
- semaphores 22
- send 19, 25, 29
- serial driver 13
- serial ports 5, 6, 7
- setdebug 29
- SetEOF 12
- SetFileInfo 10

setgid 29
SetGNOQuitRec 30
SetInGlobals 15
SetInputDevice 13
setjmp 23
SetLevel 10
SetMark 12
SetOutputDevice 13
setpgrp 29
SetPrefix 9, 10
setsystemvector 29
settpgrp 20, 29
setuid 29
shared information 21
short options 32
SIGALRM 19, 22
sigblock 29
SIGCHLD 18, 23
SIGCONT 23
SIGHUP 28
SIGINT 20, 22
SIGKILL 22, 23
signal 18, 21, 22, 23, 28, 29
signal handler 22, 23, 24
signal.h 22
sigpause 19, 29
SIGPIPE 22, 25, 31
sigsetmask 29
SIGSTOP 18, 23
SIGTERM 23
SIGTSTP 18, 20, 23
SIGTTIN 18, 20, 23
SIGTTOU 18, 23
SIGUSR 23, 24
SIGUSR1 23
SIGUSR2 23
slave 26, 27, 28
slot-dependence 13
slotNum 13
spooler 23
ssignal 21, 29
Stack 6, 30
stack frames 6
stack pointer 23
stack segment 6
stack space 17
standard loops 6
stat 29
static 6
stderr 5
stdin 5
stdio library 22
stdout 5
string manipulation 6
struct fdentry 19
struct ftable 19
SUSPENDED 18, 21
swait 21, 29
SYS.RESOURCES 15
SYSCMND 12
SysFailMgr 15
System 6.0.1 5
system call 17, 21, 23, 29, 31
system calls 29
system panic 15, 31
System Software 5.0.4 10
tcnewpgrp 20, 29
tctpgrp 20, 29
terminal 5, 6, 26
TextTools 5, 7, 11, 12, 13, 15, 16
ticks 18
time slice 17
times 29
Tool locator 5
ToolBox 5, 9, 13, 22
~ 10
ToolBox call 17, 23, 24, 31
toolerror 5
Tools 6
toOut 27
TTY 11, 18, 23, 26
tty.config 19
ttyID 18
UNIX 7, 10, 11, 16, 17, 22, 26, 32
userID 17
variable 6
virtual memory 22
wait 18, 19, 23, 29, 31
WAITING 18
WAITSIGCH 18
while 24
Window Manager 23
Write 11
writeln 22

NAME

intro - introduction to commands and application programs

DESCRIPTION

This section describes, in alphabetical order, commands available for GNO. Certain distinctions of purpose are made in the headings. For example, **BUILT-IN UTILITIES** are those commands which are part of the GNO shell and not contained in stand-alone files.

Manual Page Command Syntax

Unless otherwise noted, commands described in the **SYNOPSIS** section of a manual page accept options and other arguments according to the following syntax and should be interpreted as explained below:

name [- option ...] [cmdarg ...]

where:

[] Surround an option or cmdarg that is not required.

... Indicates multiple occurrences of the option or cmdarg.

name The name of an executable file.

option

(Almost always preceded by a "-".)

noargletter ... or,
argletter optarg [...]

noargletter

A single letter representing an option without an option-argument. Note that more than one noargletter option can be grouped after one "-".

argletter

A single letter representing an option requiring an option-argument.

optarg An option-argument (character string) satisfying a preceding argletter. Note that groups of optargs following an argletter must be separated by white space and quoted.

cmdarg

Path name (or other command argument) not beginning with a "-", or "-" by itself indicating the standard input.

SEE ALSO

GNO Shell User's Manual

NAME

`binprint` - dump binary files in ascii/hex format

SYNOPSIS

`binprint` [-c columns] [*filename*]

DESCRIPTION

`binprint` takes binary data and formats it as a sequence of ascii data and hex values that represent the binary data. The format of the output is very similar to that produced by the IIGS Monitor and the NiftyList utility.

Two columns of output are produced. The first column is the hex representation of the data. The second column is the ascii representation of the data. If the particular byte being printed is a non-printable ASCII character, it is printed as a '.'.

If the -c option is specified, the number following it is used to determine the number of columns (of bytes) per line. The default is 16.

If the filename is not specified, input is taken from standard input.

BUGS

`binprint` is slow.

AUTHOR

`binprint` was written by Derek Taubert for GNO/ME.

NAME

`cal` - display a calendar

SYNOPSIS

`cal` [[*month*] *year*]

DESCRIPTION

`cal` displays a calendar for the specified year. If a month is also specified, a calendar for that month only is displayed. If neither is specified, a calendar for the present month is printed.

year can be between 1 and 9999. Be aware that ``cal 78'` refers to the early Christian era, not the 20th century. Also, the year is always considered to start in January, even though this is historically naive.

month is a number between 1 and 12.

The calendar produced is that for England and her colonies.

Try September 1752.

NAME

cat - concatenate and print files

SYNOPSIS

cat [*-benstuv*] [file ...]

DESCRIPTION

The **cat** utility reads files sequentially, writing them to the standard output. The file operands are processed in command line order. A single dash represents standard input.

The options are as follows:

- b Implies the *-n* option but doesn't number blank lines.
- e Implies the *-v* option, and displays a dollar sign (\$) at the end of each line as well.
- n Number the output lines, starting at 1.
- s Squeeze multiple adjacent empty lines, causing the output to be single spaced.
- t Implies the *-v* option, and displays tab characters as [^I] as well.
- u The *-u* option guarantees that the output is unbuffered.
- v Displays non-printing characters so they are visible. Control characters print line [^X] for control-X; the delete character (octal 0177) prints as [^?] Non-ascii characters (with the high bit set) are printed as [M-] (for meta) followed by the character for the low 7 bits.

The **cat** utility exits 0 on success, and >0 if an error occurs.

BUGS

Because of the shell language mechanism used to perform output redirection, the command "cat file1 file 2 > file1" will cause the original data in file1 to be destroyed! Use "cat file2 >> file1" instead.

SEE ALSO

head(1), **more(1)**, **tail(1)**, Rob Pike *"UNIX Style, or cat -v Considered Harmful"*, *"USENIX Summer Conference Proceedings"* (1983)

HISTORY

A **cat** command appeared in Version 6 AT&T UNIX.

NAME

center - Center text on terminal

SYNOPSIS

center [columns] [file]

DESCRIPTION

Center is used to center lines of text either fed from stdin, or from the specified file.

One may pipe input in to it from the command line, or launch it by itself.

The commands are as follows:

columns	How many columns should be considered when centering the text. Defaults to 80 columns.
file	Specifies the file to open for centering. Defaults to stdin.

FILES

center

AUTHOR

Written by Marek Pawlowski. Source code in Public Domain. Contact author for redistribution rights, or inclusion in a software package. Munge at will. Credit to Marek Pawlowski must be retained in modified source code. Preview of modifications appreciated. Contact Marek Pawlowski at marekp@pnet91.cts.com, marekp@cerf.net.

NAME

chmod - Modify file permission flags.

SYNOPSIS

chmod [*vV*] [*octnum*][*+-=*][*rwnbi*] *file*...

DESCRIPTION

chmod is a program which modifies the permission flags of a series of files. It will modify the read, write, destroy, rename, backup, and invisible flags of the files selected.

OPTIONS

- v verbose. Provides debugging information.
- V version. Prints out the version number.

USAGE

- remove a permission flag.
- + add a permission flag.
- = add permission flag, and clear all other flags.

PERMISSIONS

- r* read permission.
- w* write permission.
- d* destroy permission.
- n* rename permission.
- b* backup needed flag.
- i* invisible flag.

EXAMPLES

- `Lock' a file: % **chmod -wdn** foo
- `Unlock' a file: % **chmod +wdn** foo

BUGS

Currently octal mode is only guaranteed to work with the ProDOS filesystem, as the **chmod()** call supplied with Orca/C doesn't seem to work well with other FSTs. Many of the standard Unix permission flags are not implemented, as the ProDOS filesystem does not support these permissions. Among unsupported permissions are separate sets of flags for user, group, and world. As well, the -x flag is not supported - if you wish to create an executable shell script, use **chtyp** instead.

SEE ALSO

ls(1), **chtyp(1)**

AUTHOR

James Brookes
bb252@cleveland.freenet.edu
jamesb@cscihp.ecst.csuchico.edu

NAME

chtyp - change file and aux types

SYNOPSIS

chtyp { [-t filetype] [-a auxtype] } | { -l lang } file...

DESCRIPTION

chtyp is used to change the file types and aux types of the specified file(s).

type is one of:

- a decimal number [66]
- a hexadecimal number preceeded by a \$ [\$42]
- an official Apple mnemonic [FTD]

and auxtype is either:

- a decimal number [64222]
- a hexadecimal number preceeded by a \$ [\$FADE]

lang is one of:

- | | |
|-----------|-------------------------|
| CC | ORCA/C |
| ASM65816 | ORCA/M or APW Assembler |
| IBASIC | ORCA/Integer Basic |
| LINK | ZapLink |
| APWC | APW C |
| PASCAL | ORCA/Pascal |
| REZ | Apple Resource Tool |
| EXEC | Shell Script file |
| TMLPASCAL | TML Pascal |

If the -l is used, the -t and -a options cannot be used.

ERRORS

If **chtyp** is interrupted with a signal (SIGINT, SIGTERM, etc.) the program aborts with a message telling what signal caused the termination.

If some other error occurs, **chtyp** aborts with an error message.

BUGS

Note that when giving hexadecimal arguments to **chtyp**, you must quote any '\$' characters with a \. For example,

```
chtyp -t \$50 -a \$8002 teach.file
```

Additional language stamps can only be added by modifying the source code.

AUTHOR

Original version by Greg Thompson.

NAME

cmp - perform a byte-by-byte comparison of two files

SYNOPSIS

cmp [**-ls**] *filename1 filename2* [*skip1*] [*skip2*]

DESCRIPTION

cmp compares *filename1* and *filename2*. If *filename1* is '-', the standard input is used. With no options, **cmp** makes no comment if the files are the same; if they differ, it reports the byte and line number at which the difference occurred, or, that one file is an initial subsequence of the other.

skip1 and *skip2* are initial byte offsets into *filename1* and *filename2* respectively, and may be either octal or decimal; a leading **0** denotes octal.

OPTIONS

- l** Print the byte number (in decimal) and the differing bytes (in octal) for all differences between the two files.
- s** Silent. Print nothing for differing files; set exit codes only.

SEE ALSO

diff(1)

DIAGNOSTICS

Exit code **0** is returned for identical files, **1** for different files, and **2** for an inaccessible or missing argument, or a system error.

NAME

compress, uncompress, zcat - compress and expand data

SYNOPSIS

compress [-cCdDf?hkKvV][-b maxbits][-Iinpath][-Ooutpath][filenames...]

uncompress [-fCcVvKk?h][-Iinpath][-Ooutpath][filenames...]

zcat [-CvV?h][-Iinpath][-Ooutpath][filenames...]

-V	print Version
-d	decompress input (default is compress)
-v	verbose
-f	force overwrite of output file (default = off)
-n	no header: useful to uncompress old files
-c	write all output to stdout (default = off)
-C	generate output compatible with compress 2.0
-k	%s input file (default = keep)
-K	%s output file on error (default = kill)
-b maxbits	default = 16 bits
-I pathname	infile path = none
-O pathname	outfile path = none
-? -h	help, print full usage message

DESCRIPTION

Compress reduces the size of the named files using adaptive Lempel-Ziv coding. Whenever possible, each file is replaced by one with the extension **.Z**, while keeping the same ownership modes, access and modification times. If no files are specified, the standard input is compressed to the standard output. Compressed files can be restored to their original form using **uncompress** or **zcat**.

The **-f** option will force compression of *name*. This is useful for compressing an entire directory, even if some of the files do not actually shrink. If **-f** is not given and **compress** is run in the foreground, the user is prompted as to whether an existing file should be overwritten.

The **-c** option makes **compress/uncompress** write to the standard output; no files are changed. The nondestructive behavior of **zcat** is identical to that of **uncompress -c**.

Compress uses the modified Lempel-Ziv algorithm popularized in "A Technique for High Performance Data Compression", Terry A. Welch, "IEEE Computer," vol. 17, no. 6 (June 1984), pp. 8-19. Common substrings in the file are first replaced by 9-bit codes 257 and up. When code 512 is reached, the algorithm switches to 10-bit codes and continues to use more bits until the limit specified by the **-b** flag is reached (default 16). Bits must be between 9 and 16. The default can be changed in the source to allow **compress** to be run on a smaller machine.

After the bits limit is attained, **compress** periodically checks the compression ratio. If it is increasing, **compress** continues to use the existing code dictionary. However, if the compression ratio decreases, **compress** discards the table of substrings and rebuilds it from scratch. This allows the algorithm to adapt to the next "block" of the file.

Note that the **-b** flag is omitted for **uncompress**, since the bits parameter specified during compression is encoded within the output, along with a magic number to ensure that neither decompression of random data nor recompression of compressed data is attempted.

The amount of compression obtained depends on the size of the input, the number of bits per code, and the distribution of common substrings. Typically, text such as source code or English is reduced by 50-60%. Compression is generally much faster compressing, but the output is not as small as freeze.

Under the `-v` option, a message is printed yielding the percentage of reduction for each file compressed.

If the `-V` option is specified, the current version and compile options are printed on stderr.

RETURN VALUE

Exit status is normally 0; if the last file is larger after (attempted) compression, the status is 2; if an error occurs, exit status is 1.

SEE ALSO

`freeze(1)`

DIAGNOSTICS

Usage: `compress [-dfvcV] [-b maxbits] [file ...]`

Invalid options were specified on the command line.

Missing maxbits

Maxbits must follow `-b`.

file : not in compressed format

The file specified to uncompress has not been compressed.

file : compressed with bits, can only handle yy bits

File was compressed by a program that could deal with more bits than the compress code on this machine. Recompress the file with smaller bits.

file : already has .Z suffix -- no change

The file is assumed to be already compressed. Rename the file and try again.

file : filename too long to tack on .Z

The file cannot be compressed because its name is longer than 12 characters. Rename and try again. This message does not occur on BSD systems.

file already exists; do you wish to overwrite (y or n)?

Respond "y" if you want the output file to be replaced; "n" if not.

Compression: "xx.xx%"

Percentage of the input saved by compression. (Relevant only for `-v`.)

-- not a regular file: unchanged

When the input file is not a regular file, (e.g. a directory), it is left unaltered.

-- file unchanged

No savings is achieved by compression. The input remains virgin.

BUGS

Although compressed files are compatible between machines with large memory, `-b 12` should be used for file transfer to architectures with a small process data space (64KB or less; as exhibited by the DEC PDP series, the Intel 80286, etc.)

NAME

conv - convert file formats

SYNOPSIS

conv -*convspec* *file1* ...

DESCRIPTION

conv converts files between various formats. *convspec* is a specification detailing the type of *file1* and the type to convert it to.

-*crlf* convert line terminators from CR (Apple) to LF (Unix).

-*lfcrlf* convert line terminators from LF (Unix) to CR (Apple).

-*detab spacing* translate tabs to spaces, using tabs every *spacing* characters. A smart algorithm is used which only inserts enough spaces to move to the next tab stop. *spacing* is an integer less than 20.

-*0001* converts all 0x00 bytes to 0x01 for using Macintosh sound files on the IIgs.

NOTES

conv is very quick on all the conversions except -*detab* (speed approaches 30K/sec). **conv** works under the Orca shell also, and supports the Orca method of wildcards. Look at the code to see how nasty this makes programs.

SEE ALSO

cat(1), **more**(1), **tr**(1)

AUTHOR

conv was written by Greg Thompson for GNO/ME.

NAME

du - Display disk usage statistics

SYNOPSIS

du [*-aksx*] *pathname* ...

DESCRIPTION

The **du** utility displays the block usage of files in the current directory or for the entire tree of a given *pathname*.

The options are as follows:

- a** Generate an entry for each file.
- k** By default, **du** displays the number of blocks as returned by the **stat(2)** system call, i.e. 512-byte blocks. If the **-k** flag is specified, the number displayed is the number of 1024-byte blocks with partial blocks rounded up.
- s** Generate only the grand total. If neither **-a** or **-s** are specified, an entry is generated for each directory only.
- x** Don't traverse any mount points.

Files having multiple hard links are counted (and displayed) a single time per **du** execution.

SEE ALSO

df(1)

BUGS

The Apple IIGS does not have the concept of mount points, and thus the **-x** option is useless.

HISTORY

A **du** command appeared in Version 6 AT&T Unix.

NAME

eps - display extended process status information.

SYNOPSIS

eps [-anlw] [-t *tty*] [-u *user*]

DESCRIPTION

eps is an extended **ps** command which displays more information than the **gsh** builtin **ps**.

USAGE

- a Show all processes; normally **eps** limits the processes displayed to those that are owned by the current user.
- n Show username instead of userID, which is default.
- l Long list. This includes PPID (parent's PID), MMID (Memory Manager ID) and a longer time field.
- w Wider list. A single *w* results in a 132 column wide listing, and two results in the whole command line being displayed. Normally the command line will be truncated to either 80 (default) or 132 (-w) columns.
- t *tty* Display only those processes that are owned by *tty*.
- u *user* Display only those processes that are owned by *user*.

SEE ALSO

GNO Shell Reference Manual, **parent(1)**

AUTHOR

James Brookes
bb252@cleveland.freenet.edu
jamesb@cscihp.ecst.csuchico.edu

NAME

fold - fold long lines for finite width output device

SYNOPSIS

fold [-w *width*]

DESCRIPTION

Fold is a filter which folds the contents of the specified files, or the standard input if no files are specified, breaking the lines to have maximum of 80 characters.

The options are as follows:

-w *width*

Specifies a line width to use instead of the default 80 characters. *Width* should be a multiple of 8 if tabs are present, or the tabs should be expanded using **conv(1)** before using **fold**.

SEE ALSO

conv(1)

BUGS

If underlining is present it may be messed up by folding.

NAME

getvers,setvers - manipulate rVersion resources in executable files

SYNOPSIS

getvers *filename*

setvers *file* '*string1 ~ string2*' [*country*] *vmajrev.minrev.bugrev*

DESCRIPTION

getvers accepts as input the name of an executable file, and prints the version information stored in the rVersion resource of the file. If no rVersion resource is present it will abort with the error 'This file has no rVersion resource'.

To add information to the rVersion resource, **setvers** is used. The rVersion format allows for two strings of up to 255 characters, although it is suggested that for this use you keep each field shorter than 80 characters.

string1 is separated from *string2* by a ~ (tilde) character, and both strings should be enclosed in single quotes. *string1* is required to be the name of the program. Any '_' character in *string2* will be interpreted as a carriage return. When using GNO, make sure to quote the single quotes and the tilde with backslashes.

The optional field *country* (no spaces allowed) allows you to set the country field of the rVersion resource. The last parameter is the current revision number of the program in the format **majrev . minrev . bugrev**, where **majrev** is a single or double digit number from 00 to 99, and **minrev** and **bugrev** are single digit numbers from zero to nine.

COUNTRIES

Valid Countries/Regions (case IS sensitive)

Arabia	Iceland
Australia	Israel
Belgium/Luxembourg	Italy
Bosnia/Herzegovina	Japan
Britian	Korea
China	Malta
Cyprus	Netherlands
Denmark	Norway
Finland	Portugal
France	Spain
FrenchCanadian	Sweden
FrenchSwiss	Taiwan
GermanSwiss	Thailand
Germany	Turkey
Greece	UnitedStates

EXAMPLES

Set the version of program 'chmod' to read:

```
chmod v01.0.0
James Brookes
jamesb@cscihp.ecst.csuchico.edu
Country: United States
```

in Orca:


```
# setvers chmod 'chmod~James Brookes_jamesb@cscihp.ecst.csuchico.edu' v01.0.0  
in GNO:
```

```
% setvers chmod \'chmod\\~James Brookes_jamesb@cscihp.ecst.csuchico.edu\'  
v01.0.0
```

CAVEATS

If an rVersion resource already exists, it will be overwritten and replaced with the new one. Other resources will be unaffected.

BUGS

Little crawly things, also known as insects.

AUTHOR

Ian Schmidt - Two Meg Software (irsman@iastate.edu)

NAME

grep, egrep, fgrep - search a file for a string or regular expression

SYNOPSIS

```
grep [ -bchilnsvw ] [ -e expression ] [ filename ... ]
egrep [ -bchilnsv ] [ -e expression ] [ -f filename ] [ expression ] [ filename ... ]
fgrep [ -bchilnsvx ] [ -e string ] [ -f filename ] [ string ] [ filename ... ]
```

DESCRIPTION

Commands of the **grep** family search the input filenames (the standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. **grep** patterns are limited regular expressions in the style of **ed**(1). **egrep** patterns are full regular expressions including alternation. **fgrep** patterns are fixed strings - no regular expression metacharacters are supported, and as a result **fgrep** is generally an order of magnitude faster than the other versions of **grep**.

Take care when using the characters '\$', '*', '[', '^', '|', '(', ')', and '\ ' in the *expression*, as these characters are also meaningful to the shell. It is safest to enclose the entire *expression* argument in single quotes `...'`

When any of the **grep** utilities is applied to more than one input file, the name of the file is displayed preceding each line which matches the pattern. The filename is not displayed when processing a single file, so if you actually want the filename to appear, use **.null** as a second file in the list.

OPTIONS

- b** Precede each line by the block number on which it was found. This is sometimes useful in locating disk block numbers by context.
- c** Display a count of matching lines rather than displaying the lines which match.
- h** Do not display filenames.
- i** Ignore the case of letters in making comparisons - that is, upper and lower case are considered identical.
- l** List only the names of files with matching lines (once) separated by NEWLINE characters.
- n** Precede each line by its relative line number in the file.
- s** Work silently, that is, display nothing except error messages. This is useful for checking the error status.
- v** Invert the search to only display lines that do not match.
- w** Search for the expression as a word as if surrounded by \< and \>. This applies to **grep** only.
- x** Display only those lines which match exactly - that is, only lines which match in their entirety. This applies to **fgrep** only.
- e *expression***
Same as a simple *expression* argument, but useful when the *expression* begins with a `-'.

-e *string*

For **fgrep** the argument is a literal character *string*.

-f *filename*

Take the regular expression (**egrep**) or a list of strings separated by NEWLINE (**fgrep**) from *filename*.

REGULAR EXPRESSIONS

The following one-character regular expressions match a single character:

- c** An ordinary character (not one of the special characters discussed below) is a one-character regular expression that matches that character.
- \ c** A backslash (\) followed by any special character is a one-character regular expression that matches the special character itself. The special characters are:
 - `\.`, `\*`, `\[`, and `\` (period, asterisk, left square bracket, and backslash, respectively), which are always special, except when they appear within square brackets (`[]`).
 - `^` (caret or circumflex), which is special at the beginning of an entire regular expression, or when it immediately follows the left of a pair of square brackets (`[]`).
 - `$` (currency symbol), which is special at the end of an entire regular expression.

A backslash followed by one of `<`, `>`, `(`, `)`, `{`, or `}`, represents a special operator in the regular expression; see below.

- A `.` (period) is a one-character regular expression that matches any character except NEWLINE.

[*string*]

A non-empty string of characters enclosed in square brackets is a one-character regular expression that matches any one character in that string. If, however, the first character of the string is a `^` (a circumflex or caret), the one-character regular expression matches any character except NEWLINE and the remaining characters in the string. The `^` has this special meaning only if it occurs first in the string. The `-` (minus) may be used to indicate a range of consecutive ASCII characters; for example, `[0-9]` is equivalent to `[0123456789]`. The `-` loses this special meaning if it occurs first (after an initial `^`, if any) or last in the string. The `]` (right square bracket) does not terminate such a string when it is the first character within it (after an initial `^`, if any); that is, `[a-f]` matches either `]` (a right square bracket) or one of the letters a through f inclusive. The four characters `.`, `*`, `[`, and `\` stand for themselves within such a string of characters.

The following rules may be used to construct regular expressions:

- *** A one-character regular expression followed by `*` (an asterisk) is a regular expression that matches zero or more occurrences of the one-character regular expression. If there is any choice, the longest leftmost string that permits a match is chosen.

\(and\)

A regular expression enclosed between the character sequences `\(` and `\)` matches whatever the unadorned regular expression matches. This applies only to **grep**.

- \n** The expression `\n` matches the same string of characters as was matched by an expression enclosed between `\(` and `\)` earlier in the same regular expression. Here *n* is a digit; the sub-expression specified is that beginning with the *n*th occurrence of `\(` counting from the left. For example, the expression `^\(.*\)\1$` matches a line consisting of two repeated appearances of the same string.

Concatenation

The concatenation of regular expressions is a regular expression that matches the concatenation of the strings matched by each component of the regular expression.

- \<** The sequence `\<` in a regular expression constrains the one-character regular expression immediately following it only to match something at the beginning of a word; that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.
- \>** The sequence `\>` in a regular expression constrains the one-character regular expression immediately following it only to match something at the end of a word; that is, either at the end of a line, or just before a character which is neither a letter, digit, nor underline.

\{ m \}
\{ m , \}
\{ m , n \}

A regular expression followed by `\{ m \}`, `\{ m , \}`, or `\{ m , n \}` matches a range of occurrences of the regular expression. The values of *m* and *n* must be non-negative integers less than 256; `\{ m \}` matches exactly *m* occurrences; `\{ m , \}` matches at least *m* occurrences; `\{ m , n \}` matches any number of occurrences between *m* and *n* inclusive. Whenever a choice exists, the regular expression matches as many occurrences as possible.

- ^** A circumflex or caret (^) at the beginning of an entire regular expression constrains that regular expression to match an initial segment of a line.
- \$** A currency symbol (\$) at the end of an entire regular expression constrains that regular expression to match a final segment of a line.

The construction

example% `^` entire regular expression `$`

constrains the entire regular expression to match the entire line.

`egrep` accepts regular expressions of the same sort `grep` does, except for `\(`, `\)`, `\n`, `\<`, `\>`, `\{`, and `\}`, with the addition of:

- *** A regular expression (not just a one-character regular expression) followed by `*` (an asterisk) is a regular expression that matches zero or more occurrences of the one-character regular expression. If there is any choice, the longest leftmost string that permits a match is chosen.
- +** A regular expression followed by `+` (a plus sign) is a regular expression that matches one or more occurrences of the one-character regular expression. If there is any choice, the longest leftmost string that permits a match is chosen.

- ? A regular expression followed by '?' (a question mark) is a regular expression that matches zero or one occurrences of the one-character regular expression. If there is any choice, the longest leftmost string that permits a match is chosen.
- | Alternation: two regular expressions separated by '|' or NEWLINE match either a match for the first or a match for the second.
- () A regular expression enclosed in parentheses matches a match for the regular expression.

The order of precedence of operators at the same parenthesis level is '[']' (character classes), then '*' '+' '?' (closures), then concatenation, then '|' (alternation) and NEWLINE.

EXAMPLES

Search a file for a fixed string using **fgrep**:

```
example% fgrep intro /usr/share/man/man3/*.3*
```

Look for character classes using **grep**:

```
example% grep '[1-8]([CJMSNX])' /usr/share/man/man1/*.1
```

Look for alternative patterns using **egrep**:

```
example% egrep '(Sally|Fred) (Smith|Jones|Parker)' telephone.list
```

To get the filename displayed when only processing a single file, use **.null** as the second file in the list:

```
example% grep 'Sally Parker' telephone.list /dev/null
```

FILES

.null

SEE ALSO

awk(1), **gsh(1)**, **vi(1)**, **sed(1)**

BUGS

Lines are limited to 1024 characters by **grep**; longer lines are truncated.

The combination of **-l** and **-v** options does not produce a list of files in which a regular expression is not found. To get such a list, use the C shell construct (Note: this is NOT the same as **gsh**, which does not support such programming).

```
foreach filename (*)
    if (`grep " re " $ filename | wc -l` == 0) echo $ filename
end
```

Ideally there should be only one **grep**.

DIAGNOSTICS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files.

NAME

head - give first few lines

SYNOPSIS

head [-*count*] [*file* ...]

DESCRIPTION

This filter gives the first *count* lines of each of the specified files, or of the standard input. If *count* is omitted it defaults to 10.

SEE ALSO

tail(1)

HISTORY

head appeared in 3 BSD.

NAME

ls - list contents of directory

SYNOPSIS

ls [**-acdfilqrstu1ACLFR**] *name* ...

DESCRIPTION

For each directory argument, **ls** lists the contents of the directory; for each file argument, **ls** repeats its name and any other information requested. By default, the output is sorted alphabetically. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments are processed before directories and their contents.

There are a large number of options:

- l** List in long format, giving mode, number of links, owner, size in bytes, and time of last modification for each file. If the file is a special file the size field will instead contain the major and minor device numbers. If the file is a symbolic link the pathname of the linked-to file is printed preceded by ```->''`.
- t** Sort by time modified (latest first) instead of by name.
- a** List all entries; in the absence of this option, entries whose names begin with a period (.) or whose GS/OS 'invisible' flag is set are not listed.
- A** List all entries except for the current directory (.) and the parent directory (..).
- s** Give size in kilobytes of each file.
- d** If argument is a directory, list only its name; often used with **-l** to get the status of a directory.
- L** If argument is a symbolic link, list the file or directory the link references rather than the link itself. Note that if the link references a directory the link is always followed, unless the **-l** option is used.
- r** Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.
- u** Use time of last access instead of last modification for sorting (with the **-t** option) and/or printing (with the **-l** option).
- c** Use time of file creation for sorting or printing.
- i** For each file, print the i-number in the first column of the report.
- f** Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off **-l**, **-t**, **-s**, and **-r**, and turns on **-a**; the order is the order in which entries appear in the directory.
- F** Cause directories to be marked with a trailing ``/'`, sockets with a trailing ``='`, executable files with a trailing ``*'`, and symbolic links to files with a trailing ``@'`. Symbolic links to directories are marked with a trailing ``/'`, unless the **-l** option is also used.
- R** recursively list subdirectories encountered.

- l force one entry per line output format; this is the default when output is not to a terminal.
- C force multi-column output; this is the default when output is to a terminal.
- q force printing of non-graphic characters in file names as the character '?'; this is the default when output is to a terminal.
- n Causes **ls** to not sort files; this is useful when organizing libraries in alphabetical order for ORCA languages.

The mode printed under the -l option contains 11 characters which are interpreted as follows: the first character is

- e if the entry has a resource fork,
- d if the entry is a directory;
- b if the entry is a block-type special file;
- c if the entry is a character-type special file;
- l if the entry is a symbolic link;
- s if the entry is a socket, or
- if the entry is a plain file.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to owner permissions; the next refers to permissions to others in the same user-group; and the last to all others. Within each set the three characters indicate permission respectively to read, to write, or to execute the file as a program. For a directory, 'execute' permission is interpreted to mean permission to search the directory. The permissions are indicated as follows:

- i if the file is invisible
- d if the file can be deleted
- r if the file is readable;
- w if the file is writable;
- x if the file is executable;
- if the indicated permission is not granted.

The group-execute permission character is given as s if the file has the set-group-id bit set; likewise the user-execute permission character is given as S if the file has the set-user-id bit set.

When the sizes of the files in a directory are listed, a total count of blocks, including indirect blocks is printed.

BUGS

The output device is assumed to be 80 columns wide.

GNO and GS/OS do not currently support links, user/group permissions, the concept of 'i-numbers', or 'special' files; thus, ls options that deal with these are ignored.

NAME

`lseg` - list segments in an OMF file

SYNOPSIS

`lseg` file

DESCRIPTION

`lseg` lists the segments of an OMF executable file. While it can list the segments in an intermediate object file, the information isn't as useful.

`lseg` is intended for discovering the location of stack segments in existing applications (for editing to smaller sizes), as an aid in determining how to segment large C files whose segments exceed the bank size, and for deciding which segments to recombine after excessive segmentation.

AUTHOR

Jawaid Bazyar for GNO/ME.

BUGS

Doesn't detect non-OMF files, and thus can get very confused if you do "`lseg *`" and one of the files chosen isn't an OMF file. Usually the only way to terminate is to interrupt the program with `^C`.

NAME

make - build a program according to a program definition file

SYNOPSIS

make [-d] [-p] [-s] inputFile

DESCRIPTION

make is a program maintenance utility that aids in creating programs from multiple source files.

inputFile is the name of the Program Description File. If absent, make defaults to using 'makefile' as the PDF. make's options are as follows:

- d Display the modification date and time as each file is checked.
- p Operate in programmer/debug mode.
- s Operate in silent mode.

The logical definition of each of the PDF parameters follows:

```
# Comments begin with a pound sign "#".
#
TargetFile: DependentFile1 DependentFile2 \ # Continuation line
           DependentFileN
           ShellCommand1
           ShellCommand2
           .
           .
           ShellCommandN
```

MAKE PARAMETERS

Comments can be placed anywhere in the PDF file. If a pound sign is in column one, the entire line is treated as a comment. Anything following the pound is ignored by make. Comments may also be placed on parameter lines containing Target/Dependent file parameters.

The TargetFile parameter must start in column one and ends with a colon ":". It can be a full path name, partial path name or file in the current directory. This file is usually an object type file created by a compiler. Dependent file parameter(s) follow the semicolon and are separated by a space, a comma or both. Dependent file parameters are usually CC, PAS, ASM or some type of included SRC file. Essentially, TargetFile is the result of compiling the dependent source files.

make obtains the modification date and time of the Target file and then compares it to each of the dependent file(s) moving left to right. If one of the dependent files has a date and time later than the Target file, the subsequent ShellCommands are executed. Target/Dependent parameters may be continued by placing a reverse slash "\" on the line following a dependent file parameter. make will interpret the next line as dependent file parameters which may also be continued and so on. There should not be any blank lines between the continued line and next line.

ShellCommands must contain a space in column one to differentiate them from Target/Dependent file parameters. If make determines that a Target file needs to be recreated, the ShellCommands following the Target/Dependent file parameters are passed, as is, to the shell interpreter. The commands must be valid shell commands. ShellCommands are executed until a blank line is encountered or an error occurs during the execution of the last command. If an error occurs, make terminates without reading the remaining PDF parameters. If a blank line is read, make returns to Target/Dependent search mode.

MAKE Example


```
#
# File: Example.Make - A MAKE example PDF #

Menu.Root: Menu.CC # Contains menu related routines
             compile menu.cc keep=menu

Window.Root: Window.CC # Contains window handling routines
             compile window.cc keep=window

Misc.Root: Misc.CC Misc.h # Contains miscellaneous program routines
             compile misc.cc keep=misc

Main.Root: Main.CC Include/Main.h Include/Misc.h # Main program
             compile main.cc keep=main

MyProg: Menu.Root Window.Root Misc.Root \ # re-link required?
        Main.Root                       # check all dependent files
        link main misc menu window keep=myprog # Shell comment
        chtyp -t sl6 myprog                # change type to S16

# End of PDF
```

The example PDF shows a program that is made up of 4 source files. The Main and Misc routines are also dependent on include files containing information that if changed would force a re-compile of that module.

One thing to keep in mind is the order of Target/Dependent parameters can be important. make examines the PDF from the top down. If a file is modified due to a command later in the PDF, make will not return to a previous Target/Dependent parameter in which that file was a dependent file.

AUTHOR

Original make util written by Larry Agle for ORCA; modifications made to make it look more like UNIX make, and misc. bug fixes by Jawaidd Bazyar.

NAME

makemake - scan C source files for dependencies and create a makefile

SYNOPSIS

makemake source.c

DESCRIPTION

makemake takes the C source file(s) as input, and scans them for dependencies (`#includes`). It does not count standard headers (`#includes` with the filename surrounded by `< >`).

The dependency information for all the source files specified is then written to a file 'makefile' in the current directory.

makemake does not create link scripts nor does it put an executable dependency into the file, as these require link information not available from the **makemake** command line.

AUTHOR

Jawaid Bazyar for GNO/ME's 'make' utility. He got tired of writing them by hand.

BUGS

makemake does not properly handle recursive or multiple includes of the same file. If the same file is included more than once, **makemake** will list it that many times in the output makefile. While multiple includes don't hurt anything, it can take more time to process the makefile. Recursive includes will, of course, hang **makemake**.

NAME

man - online manual system (Version 1.6)

SYNOPSIS

man [*section*] *manpagename*

DESCRIPTION

man is the access point to the online manual system. **man** works as a shell that calls an appropriate text formatter to format a manual page. There are currently three text formatters available for the GNO/ME system.

nroff Unix standard typesetting package

aroff formats AppleWorks GS(tm) documents

cat used to display preformatted documents

If the option section argument is specified, **man** looks specifically in that section of the manual for the manpage. This is needed in situations where there are manpages with the same name in different sections (for example, **sleep**(1) and **sleep**(3)).

ENVIRONMENT

man recognizes the following variables from the shell.

USRMAN

This variable is required. It points to the manual system root directory.

PAGER

man uses **more** as the default pager. If you wish to use a different pager, **less** perhaps, then you must set this variable to point to that pager.

Compressed Manual Pages and Links

If **man** finds a file in the manual that ends in a '.l' suffix, it takes the contents of that file as a 'link' to the actual manpage. This is useful for manpages that describe multiple commands, and prevents having multiple copies of the manpage.

Manual pages may be compressed with **compress** or **freeze**, in which case the appropriate program is called to uncompress the manual file.

SEE ALSO

more(1), **less**(1), **compress**(1), **freeze**(1).

NAME

mkdir - Makes directories

SYNOPSIS

mkdir *dirname* ...

DESCRIPTION

The **mkdir** command creates subdirectories with the *dirname*s specified. If a file with that *dirname* exists, an error is returned. *dirname* may be a full pathname, or a partial pathname, in which case the directory is created as a subdirectory of the current directory.

mkdir only creates the filename portion of the specified path. If, for instance, you do

mkdir /usr/local/bbs/foo

directory foo will only be created if all of **/usr**, **local**, and **bbs** exist.

AUTHOR

mkdir was written by James Brookes for GNO/ME.

NAME

more - text pager

SYNOPSIS

more [*file* ...]

DESCRIPTION

more allows the user to view the specified *file(s)* screen by screen or line by line. If no *files* are specified, standard input is used.

Every time **more** has displayed a screen of text, it displays a prompt

- filename (xx%) -

indicating the percentage of the file that has been viewed and its filename. If standard input is used,

' - more - '

is used as the prompt instead.

A number of key commands are available at the prompt.

q quit viewing the current file, and move to the next file (if any)

[RETURN]

display the next line of the file

[ESC]

abort **more**, including any more files that may have been specified

[SPACE]

display the next page of the file

AUTHOR

This version of more was written by Jawaid Bazyar and Derek Taubert.

NAME

passwd - set a user's login password

SYNOPSIS

passwd [-? | -v] [*username*]

DESCRIPTION

passwd changes the specified user's password. Only root is allowed to alter passwords other than his own. If the *username* is not given, the user's own login name is assumed. Users other than root must then enter the old password to verify permission to change the password. Finally, the user must type the desired new password twice to insure that no mistakes are made.

To cancel **passwd**, type CTRL-@ when asked to enter the new password.

The -? flag causes **passwd** to display a brief usage message, and the -v flag causes **passwd** to display version information.

SEE ALSO

login(1)

FILES

/etc/passwd - contains the password information

AUTHOR

Eric Shepherd
Internet uerics@mcl.mcl.ucsb
AOL Sheppy

NAME

purge - deallocate purgeable memory handles

SYNOPSIS

purge [-v]

DESCRIPTION

This program purges all memory blocks marked purgable. This is very important when using the Orca compilers and shells since certain blocks get left lying around and can (and have!) caused compiler errors. By purging memory before compiles a large percentage of strange compiler errors can be eliminated. After purging, all memory possible is then freed for usage.

-v gives a verbose listing of each handle being purged and before and after free memory statistics.

The problems mentioned above usually occur when a program has over-written one of the ORCA FastFile system's memory handles. Purging clears these handles and forces a reload from disk.

BUGS

Purging memory when non-shell applications are running could be dangerous.

NAME

rx, rc, rb, rz - Receive Files and Commands with X/Y/ZMODEM

SYNOPSIS

rz -tv
rb -tv
rc -tv file
rx -tv file
gz file ... [-v]
rzCOMMAND

DESCRIPTION

This program uses error correcting protocols to receive files over a dial-in serial port from a variety of programs running under many operating systems. It is invoked from a shell prompt manually, or automatically as a result of an "sz file ..." command given to the calling program.

This is a shareware program copyrighted by Omen Technology INC.

Rz (Receive ZMODEM) receives one or more files with the ZMODEM protocol. Pathnames are supplied by the sending program, and directories are made if necessary (and possible). Normally, the "rz" command is automatically issued by the calling ZMODEM program, but defective ZMODEM implementations may require starting **rz** manually.

Rb receives file(s) with YMODEM, accepting either standard 128 byte sectors or 1024 byte sectors (sb -k option). The user should determine when the 1024 byte block length actually improves throughput.

If True YMODEM™ (Omen Technology trademark) file information (file length, etc.) is received, the file length controls the number of bytes written to the output dataset, and the modify time and file mode (iff non zero) are set accordingly.

If True YMODEM file information is not received, slashes in the pathname are changed to underscore, and any trailing period in the pathname is eliminated. This conversion is useful for files received from CP/M and other historical systems.

Rc receives a single file with XMODEM-CRC or XMODEM-1k-CRC protocol. The user should determine when the 1024 byte block length actually improves throughput without causing problems. The user must supply the file name to both sending and receiving programs. Up to 1023 garbage characters may be added to the received file.

Rx receives a single file with XMODEM or XMODEM-1k protocol. The user should determine when the 1024 byte block length actually improves throughput without causing problems. The user must supply the file name to both sending and receiving programs. Up to 1023 garbage characters may be added to the received file.

Rz may be invoked as **rzCOMMAND** (with an optional leading - as generated by login(1)). For each received file, **rz** will pipe the file to ``COMMAND filename" where filename is the name of the transmitted file with the file contents as standard input.

Each file transfer is acknowledged when **COMMAND** exits with 0 status. A non zero exit status terminates transfers.

A typical use for this form is rzmail which calls rmail(1) to post mail to the user specified by the transmitted file name. For example, sending the file "caf" from a PC-DOS system to rzmail on a Unix system would result in the contents of the DOS file "caf" being mailed to user "caf".

The meanings of the available options are:

- tim** Change timeout to tim tenths of seconds.
- v** Verbose causes a list of file names to be appended to /tmp/rzlog . More v's generate more detailed debugging output.

DIAGNOSTICS

Exit status is as follows: 0 for successful transfers. 1 if unrecoverable errors are detected. 2 if syntax errors or file access problems are detected. 3 if the program was terminated by a caught interrupt.

SEE ALSO

sz(1).

NOTES

ZMODEM's support of XOFF/XON flow control allows proper operation in many environments that do not support XMODEM uploads. Unfortunately, not all Unix versions support input flow control. The TTY input buffering on some systems may not adequately buffer long blocks or streaming input at high speed. You should suspect this problem when you can't send data to the Unix system at high speeds using ZMODEM, YMODEM-1k or XMODEM-1k, but YMODEM with 128 byte blocks works properly.

If a program that does not properly implement the specified file transfer protocol causes **rz** to "hang" the port after a failed transfer, either wait for **rz** to time out or keyboard a dozen Ctrl-X characters.

Many programs claiming to support YMODEM only support XMODEM with 1k blocks, and they often don't get that quite right.

BUGS

This version of **rz** does not support some ZMODEM features.
The ASCII option's CR/LF to NL translation merely deletes CR's.

ZMODEM CAPABILITIES

Rz supports ZMODEM command execution (zcommand), incoming ZMODEM binary (-b), ASCII (-a), newer(-n), newer+longer(-N), protect (-p), Crash Recovery(-r), clobber (-y), match+clobber (-Y), compression(-Z), and append (-+) requests. Other options sent by the sender are ignored. The default is protect (-p) and binary (-b).

FILES

/tmp/rzlog stores debugging output generated with -vv option

NAME

script - make typescript of a terminal session

SYNOPSIS

script [**-a**] [*filename*]

DESCRIPTION

script makes a typescript of everything printed on your terminal. The typescript is written to *filename* , or appended to *filename* if the **-a** option is given. It can be sent to the line printer later with **lpr**(1). If no file name is given, the typescript is saved in the file **typescript** .

The script ends when the forked shell exits.

OPTIONS

-a Append the script to the specified file instead of writing over it.

SEE ALSO

lpr (1), **pty** (4)

BUGS

script places *everything* in the log file. This is not what the naive user expects.

NAME

sleep - suspend execution for an interval

SYNOPSIS

sleep *time*

DESCRIPTION

Sleep suspends execution for *time* seconds. It is used to execute a command after a certain amount of time as in a script:

```
sleep 105  
command
```

SEE ALSO

alarm(3C), sleep(3)

BUGS

Time must be less than 2,147,483,647 seconds.

NAME

split - split a file into pieces

SYNOPSIS

split [- *number*] [*infile* [*outfile*]]

DESCRIPTION

split reads *infile* and writes it in *number* -line pieces (default 1000) onto a set of output files (as many files as necessary). The name of the first output file is *outfile* with **aa** appended, the second file is **outfileab** , and so on lexicographically.

If no *outfile* is given, **x** is used as default (output files will be called **xaa** , **xab** , etc.).

If no *infile* is given, or if '-' is given in its stead, then the standard input file is used.

OPTIONS

- *number* Number of lines in each piece.

NAME

stty - set and view terminal options and parameters

SYNOPSIS

stty [*option ...*] [*charoption c ...*]

DESCRIPTION

If no options are specified, **stty** prints out all the current terminal option settings. *options* represent boolean flags in the terminal parameters, and are as follows:

raw	turns on RAW mode (no character or line processing)
-raw	turns off RAW mode
ehco	if in CBREAK or COOKED mode, echoes input characters
-echo	echo mode off
cbreak	turns on CBREAK mode (single character processing)
-cbreak	turns off CBREAK mode (line-at-a-time processing)

Charoptions represent variables in the terminal interface, and are as follows:

intr c	sets the interrupt character (normally ^C)
start c	sets the start character (normally ^Q)
stop c	sets the stop character (normally ^S)
eof c	sets the eof character (normally ^D)
susp c	sets the suspend character (normally ^Z)

c may be defined either as an octal number such as 003, or in control character format (^C).

SEE ALSO

tty(4)

NAME

sum - print checksum and block count of a file

SYNOPSIS

sum [*file*]

DESCRIPTION

Sum calculates and prints a 16-bit checksum for the named file, and also prints the number of blocks in the file. Stdin is used if no file names are given. **Sum** is typically used to look for corrupted files, or to validate a file communicated over some transmission line.

DIAGNOSTICS

``Read error" is indistinguishable from end of file on most devices; check the block count.

SEE ALSO

wc(1).

NOTE

Sum is pretty slow on large files when running on the GS. If anyone has a faster algorithm for computing the 16-bit checksum, I'd really appreciate seeing it.

AUTHOR

Marek Pawlowski - marekp@pnet91.cts.com

NAME

sx, sb, sz - Send Files with ZMODEM, YMODEM, or XMODEM
zcommand, zcommandi - Send COmmands with ZMODEM

SYNOPSIS

sz [-+abdefkLlNnopTtuvyYZ] file ...
sb [-dfktuv] file ...
sx [-ktuv] file
zcommand [-otv] COMMAND
zcommandi [-otv] COMMAND
sz -TT

DESCRIPTION

Sz (send ZMODEM) uses the ZMODEM, YMODEM or XMODEM error correcting protocol to send one or more files over a dial-in serial port to a variety of programs running under PC-DOS, CP/M, Unix, VMS, and other operating systems.

This is a shareware program copyrighted by Omen Technology INC.

Sz sends one or more files with ZMODEM protocol.

ZMODEM greatly simplifies file transfers compared to XMODEM. In addition to a friendly user interface, ZMODEM provides Personal Computer and other users an efficient, accurate, and robust file transfer method.

ZMODEM provides complete END-TO-END data integrity between application programs. ZMODEM's 32 bit CRC catches errors that sneak into even the most advanced networks.

Advanced file management features include AutoDownload (Automatic file Download initiated without user intervention), Display of individual and total file lengths and transmission time estimates, Crash Recovery, selective file transfers, and preservation of exact file date and length.

The -y option instructs the receiver to open the file for writing unconditionally. The -a option causes the receiver to convert Unix newlines to PC-DOS carriage returns and linefeeds.

Sb sends one or more files with YMODEM or ZMODEM protocol. The initial ZMODEM initialization is not sent. When requested by the receiver, **sb** supports YMODEM-g with "cbreak" tty mode, XON/XOFF flow control, and interrupt character set to CAN (^X). YMODEM-g increases YMODEM throughput over error free channels (direct connection, X.PC, etc.) by disabling error recovery.

On Unix systems, additional information about the file is transmitted. If the receiving program uses this information, the transmitted file length controls the exact number of bytes written to the output dataset, and the modify time and file mode are set accordingly.

Sx sends a single file with XMODEM or XMODEM-1k protocol (sometimes incorrectly called "ymodem"). The user must supply the file name to both sending and receiving programs.

If **sz** is invoked with \$SHELL set and if that variable contains the string rsh or rksh (restricted shell), **sz** operates in restricted mode. Restricted mode restricts pathnames to the current directory and PUBDIR (usually /usr/spool/uucppublic) and/or subdirectories thereof.

The fourth form sends a single COMMAND to a ZMODEM receiver for execution. Zcommand exits with the COMMAND return value. If COMMAND includes spaces or characters special to the shell, it must be quoted.

The fifth form sends a single COMMAND to a ZMODEM receiver for execution. Zcommandi exits as soon as the receiver has correctly received the command, before it is executed.

The sixth form (sz -TT) attempts to output all 256 code combinations to the terminal. If you are having difficulty sending files, this command lets you see which character codes are being eaten by the operating system.

The meanings of the available options are:

- +** Instruct the receiver to append transmitted data to an existing file (ZMODEM only).
- a** Instruct the ZMODEM receiver to convert text file format as appropriate for the receiving system. Valid only for ZMODEM.
- b (Zmodem)** Binary override: transfer file without any translation.
- c** Instruct the receiver to change the pathname if the destination file exists.
- d** Change all instances of "." to "/" in the transmitted pathname. Thus, C:omenB0000 (which is unacceptable to MSDOS or CP/M) is transmitted as C:/omenB0000. If the resultant filename has more than 8 characters in the stem, a "." is inserted to allow a total of eleven.
- e** Escape all control characters; normally only XON, XOFF, DLE, CR-@-CR, and Ctrl-X are escaped.
- f** Send Full pathname. Normally directory prefixes are stripped from the transmitted filename.
- k (X/Ymodem)** Send files using 1024 byte blocks rather than the default 128 byte blocks. 1024 byte packets speed file transfers at high bit rates. (ZMODEM streams the data for the best possible throughput.)
- L N** Use ZMODEM sub-packets of length N. A larger N ($32 \leq N \leq 1024$) gives slightly higher hroughput, a smaller N speeds error recovery. The default is 128 below 300 baud, 256 above 300 baud, or 1024 above 2400 baud.
- l N** Wait for the receiver to acknowledge correct data every N ($32 \leq N \leq 1024$) characters. This may be used to avoid network overrun when XOFF flow control is lacking.
- n (Zmodem)** Send each file if destination file does not exist. Overwrite destination file if source file is newer than the destination file.
- N (Zmodem)** Send each file if destination file does not exist. Overwrite destination file if source file is newer or longer than the destination file.
- o (Zmodem)** Disable automatic selection of 32 bit CRC.
- p (Zmodem)** Protect existing destination files by skipping transfer if the destination file exists.
- r (Zmodem)** Resume interrupted file transfer. If the source file is longer than the destination file, the transfer commences at the offset in the source file that equals the length of the destination file.
- rr** As above, but compares the files (the portion common to sender and reciever) before resuming the transfer.
- t tim** Change timeout to tim tenths of seconds.
- u** Unlink the file after successful transmission.
- w N** Limit the transmit window size to N bytes (ZMODEM).
- v** Verbose causes a list of file names to be appended to /tmp/szlog. More v's generate more output.
- y** Instruct a ZMODEM receiving program to overwrite any existing file with the same name.

- Y** Instruct a ZMODEM receiving program to overwrite any existing file with the same name, and to skip any source files that do have a file with the same pathname on the destination system.
- Z** Use ZMODEM file compression to speed file transfer.

DIAGNOSTICS

Exit status is as follows: 0 for successful transfers. 1 if unrecoverable errors are detected. 2 if syntax errors or file access problems are detected. 3 if the program was terminated by a caught interrupt.

EXAMPLE

ZMODEM File Transfer (GNO to remote system)

```
% sz -a *.c
```

This single command transfers all .c files in the current directory with conversion (-a) to end of line conventions appropriate to the receiving environment. With ZMODEM AutoDownload enabled, will automatically receive the files after performing a security check.

```
% sz -Yan *.c *.h
```

Send only the .c and .h files that exist on both systems, and are newer on the sending system than the corresponding version on the receiving system, converting Apple to UNIX text format.

SEE ALSO

rz(1).

Compile time options required for various operating systems are described in the source file.

FILES

32 bit CRC code courtesy Gary S. Brown.

sz.c, crctab.c, rbsb.c, zm.c, zmr.c, zmodem.h Unix source files

/tmp/szlog stores debugging output (sz -vv)

TESTING FEATURE

The command "sz -T file" exercises the Attn sequence error recovery by commanding errors with unterminated packets. The receiving program should complain five times about binary data packets being too long. Each time sz is interrupted, it should send a ZDATA header followed by another defective packet. If the receiver does not detect five long data packets, the Attn sequence is not interrupting the sender, and the Myattn string in sz.c must be modified.

After 5 packets, sz stops the "transfer" and prints the total number of characters "sent" (Tcount). The difference between Tcount and 5120 represents the number of characters stored in various buffers when the Attn sequence is generated.

NOTES

When using buffered modems at high speed, particular attention must be paid to flow control. The modem and Unix must agree on the flow control method. Sz on USG (SYS III/V) systems uses XON/XOFF flow control. If flow control cannot be properly set up, Try a "-w 2048" option to enforce protocol level flow control. Experiment with different window sizes for best results.

If a program that does not properly implement the specified file transfer protocol causes sb to "hang" the port after a failed transfer, either wait for sb to time out or type a dozen Ctrl-X characters.

Many programs claiming to support YMODEM only support XMODEM with 1k blocks, and they often don't get that quite right. XMODEM transfers add up to 127 garbage bytes per file. XMODEM-1k and YMODEM-1k transfers use 128 byte blocks to avoid extra padding.

YMODEM programs use the file length transmitted at the beginning of the transfer to prune the file to the correct length; this may cause problems with source files that grow during the course of the transfer. This problem does not pertain to ZMODEM transfers, which preserve the exact file length unconditionally.

Most ZMODEM options are merely passed to the receiving program; some programs do not implement all of these options.

Circular buffering and a ZMODEM sliding window should be used when input is from pipes instead of acknowledging frames each 1024 bytes. If no files can be opened, sz sends a ZMODEM command to echo a suitable complaint; perhaps it should check for the presence of at least one accessible file before getting hot and bothered.

BUGS

On at least one BSD system, sz would abnormally end if it got within a few kilobytes of the end of file. Using the "-w 8192" flag fixed the problem. The real cause is unknown, perhaps a bug in the kernel TTY output routines.

The test mode leaves a zero length file on the receiving system.

GNO/ME

The usual manner of invoking sz to send files from a IIgs is as follows:

Connect to the other computer with a term program such as TelCom GS

Start the X/Y/Zmodem receive on the other side

Get/Quit back to the GNO Shell

Type:

```
sz -v -v -b filename1 filename2 .. <.ttyp >.ttyp
```

You may put this operation in the background of course. Tests have shown no data loss up to 9600 baud in background operation.

The -b option ensures binary mode. You must use this if you're sending a ShrinkIt archive or other binary file. For plain text files you can leave off the -b.

NAME

tar - extract and view tape archives

SYNOPSIS

tar [-]{x|t}f[v] *archive*

DESCRIPTION

tar lists the contents of and extracts files from UNIX tape archives (*.tar files).

Traditionally, **tar** does not require the normal '-' character to denote its arguments. The option flags are as follows:

- x Extract files from the archive
- v Verbose mode (tell what tar is doing)
- t Tell mode (list files in archive)
- f Use a file on disk instead of a tape

Since the standard IIGS filesystem is not as flexible, filename-wise, as UNIX filesystems, some pre-processing is performed on filenames created when an archive is extracted.

- If a tar filename contains a double-/ (possible under UNIX if an archive was created by specifying a directory with a trailing slash), tar converts it to a single /.
- If a filename contains non-alpha numeric characters, they are converted to periods ('.').

tar does not maintain the file protection bits from UNIX, nor does it maintain the creation and modification dates.

BUGS

- Does not create .tar archives
- Does not work with raw devices, only files
- Does not allow user to specify which files to extract from archive
- tar should use the GS/OS JudgeName call.
- The -x and -t options should be exclusive, but are not.

NAME

tee - Pipe fitting.

SYNOPSIS

tee [-ai] [*file* ...]

DESCRIPTION

The **tee** utility copies standard input to standard output, making a copy in zero or more files. The output is unbuffered.

The following options are available:

-a Append the output to the files rather than overwriting them.

-i Ignore the **SIGINT** signal.

The following operands are available:

file A pathname of an output file .

The **tee** utility takes the default action for all signals, except in the event of the **-i** option.

The **tee** utility exits 0 on success, and >0 if an error occurs.

STANDARDS

The **tee** function is expected to be POSIX p1003.2 compatible.

SEE ALSO

GNO/Shell User's Manual , **signal(2)**

NAME

time - time a command

SYNOPSIS

time command

DESCRIPTION

The **time** command lets the specified **command** execute and then outputs the amount of elapsed real time, the time spent in the operating system, and the time spent in execution of the command. Times are reported in seconds and are written to standard error.

The **time** command can be used to cause a command to be timed no matter how much CPU time it takes. For example:

```
% /bin/time cp /etc/rc /usr/bill/rc
0.1 real    0.0 user    0.0 sys
```

```
% /bin/time nroff sample1 > sample1.nroff
3.6 real    2.4 user    1.2 sys
```

This example indicates that the **cp** command used negligible amounts of user (*user*) and system time (*sys*), and had an elapsed time (*real*) of 1/10 second (0.1). The **nroff** command used 2.4 seconds of user time and 1.2 seconds of system time, and required 3.6 seconds of elapsed time.

RESTRICTIONS

Times are measured to an accuracy of 1/60 second. Thus, the sum of the user and system times can be larger than the elapsed time, but this isn't likely.

NAME

tr - translate characters

SYNOPSIS

tr [**-cds**] [*string1* [*string2*]]

DESCRIPTION

tr copies the standard input to the standard output with substitution or deletion of selected characters. The arguments *string1* and *string2* are considered sets of characters. Any input character found in *string1* is mapped into the character in the corresponding position within *string2*. When *string2* is short, it is padded to the length of *string1* by duplicating its last character.

In either string the notation '**a - b**' denotes a range of characters from **a** to **b** in increasing ASCII order. The character ****, followed by 1, 2 or 3 octal digits stands for the character whose ASCII code is given by those digits. As with the shell, the escape character ****, followed by any other character, escapes any special meaning for that character.

When *string2* is short, characters in *string1* with no corresponding character in *string2* are not translated.

In either string the following abbreviation conventions introduce ranges of characters or repeated characters into the strings.

[**a - z**]

Stands for the string of characters whose ASCII codes run from character **a** to character **z**, inclusive.

[**a * n**]

Stands for **n** repetitions of **a**. If the first digit of **n** is 0, **n** is considered octal; otherwise, **n** is taken to be decimal. A zero or missing **n** is taken to be huge; this facility is useful for padding *string2*.

OPTIONS

Any combination of the options **-c**, **-d**, or **-s** may be used:

-c Complement the set of characters in *string1* with respect to the universe of characters whose ASCII codes are 01 through 0377 octal;

-d Delete all input characters in *string1*.

-s Squeeze all strings of repeated output characters that are in *string2* to single characters.

EXAMPLE

The following example creates a list of all the words in *filename1* per line in *filename2*, where a word is taken to be a maximal string of alphabets. The second string is quoted to protect **** from the shell. 012 is the ASCII code for NEWLINE.

```
tr -cs A-Za-z '\012' < filename1 > filename2
```

SEE ALSO

conv(1), **more(1)**

BUGS

Will not handle ASCII NUL in *string1* or *string2*. **tr** always deletes NUL from input.

NAME

unshar - extracts files from shar archives

SYNOPSIS

unshar {-overwrite} {-nosort} *file1 file2 ...*

DESCRIPTION

Unshar is a utility which extracts files from the ubiquitous Unix shar archives. It has the following advantages over existing unshar utilities:

- Small and fast
- Handles many cat and sed formats
- Allows extraction of subdirectories
- Understands ./file type filenames
- Understands file continuation with >>
- Sorts file list by Subject: line
- Exits cleanly with CTRL-C

Unshar treats quotes and imbedded escape sequences intelligently and handles all the cat and sed formats the author ever seen, including sed commands which strip off more than one letter. There may be some formats it won't handle, but I've yet to find them.

Invoke **unshar** with as many archive filenames as you like. All the files in each archive will be extracted into the current directory. If a file already exists, **unshar** asks you how you want to handle it. Entering 'Y' will overwrite the existing file with the version in the archive, 'N' will skip past the file without extracting it, and 'A' will overwrite this file and any other existing files without prompting you again. Including the -o flag on the command line causes files to always be overwritten.

OPERATION

Unshar scans through each archive specified on the command line, looking for lines beginning with 'cat' or 'sed'. All other lines are ignored. In particular, 'echo' lines are not echoed. This way, you don't get a load of messages which are in any case fairly meaningless, because the operations they are describing are unsupported.

When a shar archive contains a file for which a full pathname is given (as in source/file.c for example) unshar will create whatever directories are necessary. It also strips off leading /'s and ./'s, to make filenames understandable by GS/OS.

Occasionally, a shar file distribution will contain a file too large to fit into a single shar archive (archives are typically limited to around 60K or so for transmission over Usenet). One method some archivers use to get around this is to split the large file into several smaller parts, and use the shell '>>' redirection operator to concatenate the parts together while extracting the files. In order for this to work properly, it is important that the archive files are extracted in the correct sequence; otherwise, all the pieces will get joined together in the wrong order.

To assist with this, **unshar** does a prescan over all the files listed on the command line, and checks each file for a "Subject:" line. If it finds such a line, it scans it looking for any hints as to where the file comes in the sequence. Most shar files you feed to unshar will be directly from a Usenet sources or binaries group, and will include a volume and issue reference on the subject line. If **unshar** can't find such an issue reference, it will look for a Part number and use that instead.

Unshar then extracts the archives starting with the lowest numbered file. This helps to ensure that those extra-large files are created correctly. You can tell when such a file is being created, because **unshar** says "Extending file" rather than "Unsharing file".

If for some reason you want the files to be unarchived in the order listed on the command line, you can specify the ``-n'` (nosort) switch, and no sorting will take place.

HISTORY

V1.0 First release.

V1.1 Added support for some more unusual uses of sed.
Increased speed, and reduced size slightly.
Fixed bug that truncated lines longer than 80 chars.

V1.2 Added support for sorting by Subject: line
Added support for file appending via >>
Fixed small bug in detection of disk write errors

AUTHOR

Eddy Carroll (EMAIL: ecarroll@vax1.tcd.ie)
Apple //gs port by Andy McFadden (fadden@uts.amdahl.com).

NAME

wc - display a count of lines, words and characters

SYNOPSIS

wc [**-lwc**] [*filename* ...]

DESCRIPTION

wc counts lines, words, and characters in *filename* s, or in the standard input if no filename appears. It also keeps a total count for all named files. A word is a string of characters delimited by SPACE, TAB, or NEWLINE characters.

OPTIONS

When *filename* s are specified on the command line, their names will be printed along with the counts.

The default is **-lwc** (count lines, words, and characters).

-l Count lines.
-w Count words.
-c Count characters.

EXAMPLE

```
example% wc $USRMAN/csh.1 $USRMAN/sh.1 $USRMAN/telnet.1
1876      11223      65895  /usr/share/man/man1/csh.1
674       3310      20338  /usr/share/man/man1/sh.1
260       1110      6834   /usr/share/man/man1/telnet.1
2810     15643     93067   total
example%
```

NAME

who - who is on the system

SYNOPSIS

who [who-file] [am I]

DESCRIPTION

Who, without an argument, lists the login name, terminal name, and login time for each current UNIX user.

Without an argument, **who** examines the **/etc/utmp** file to obtain its information. If a file is given, that file is examined. Typically the given file will be **/usr/adm/wtmp**, which contains a record of all the logins since it was created. Then **who** lists logins, logouts, and crashes since the creation of the wtmp file. Each login is listed with user name, terminal name (with **`/dev/'** suppressed), and date and time. When an argument is given, logouts produce a similar line without a user name. Reboots produce a line with **`x'** in the place of the device name, and a fossil time indicative of when the system went down.

With two arguments, as in **`who am I'** (and also **`who are you'**), **who** tells who you are logged in as.

FILES

/etc/utmp

SEE ALSO

getuid(2), **utmp(5)**

NAME

yes - be repetitively affirmative

SYNOPSIS

yes [*expletive*]

DESCRIPTION

Yes repeatedly outputs "y", or if *expletive* is given, that is output repeatedly. Termination is by interrupt.

NAME

intro - introduction to GNO system calls

DESCRIPTION

This section describes, in alphabetical order, the system calls and kernel interfaces available for GNO. Certain distinctions of purpose are made in the headings.

Manual Page Command Syntax

Unless otherwise noted, calls described in the **SYNOPSIS** section of a manual page accept parameters and require the

name [- option ...] [cmdarg ...]

where:

[] Surround an option or cmdarg that is not required.

... Indicates multiple occurrences of the option or cmdarg.

name The name of an executable file.

option (Almost always preceded by a "-".)
noargletter ... or,
argletter optarg [...]

noargletter A single letter representing an option without an option-argument. Note that more than one noargletter option can be grouped after one "-".

argletter A single letter representing an option requiring an option-argument.

optarg An option-argument (character string) satisfying a preceding argletter. Note that groups of optargs following an argletter must be separated by white space and quoted.

cmdarg Path name (or other command argument) not beginning with a "-", or "-" by itself indicating the standard input.

SEE ALSO

GNO/ME Kernel Reference Manual

NAME

alarm - set and reset alarm timer

SYNOPSIS

```
#include <gno/gno.h>
long int alarm(long int seconds);
```

DESCRIPTION

alarm sets the counter on the calling process' alarm timer to the value specified in *seconds*. If *seconds* is (long int) 0, the alarm timer is disabled.

When an alarm timer terminates (by counting down to 0), the calling process is sent a SIGALRM signal.

RETURN VALUE

The amount of time previously remaining in the timer is returned. No errors are possible.

SEE ALSO

signal(2), **sigpause(2)**

NAME

dup, dup2 - duplicate open file descriptors

SYNOPSIS

```
#include <gno/gno.h>
```

```
int dup(int filedes);  
int dup2(int filedes, int filedes2);
```

DESCRIPTION

Given the file descriptor of a valid open file, **dup** creates a new file descriptor that is synonymous with *filedes*. The new file descriptor is returned. The second form of the call forces an existing file descriptor *filedes2* to refer to the same file as *filedes*. If *filedes2* already refers to an open file, it is closed first.

RETURN VALUE

dup returns the new file descriptor, and **dup2** returns 0 if the calls are successful. In the event of an error, -1 is returned and **errno** is set as follows:

[EBADF]	<i>filedes</i> refers to an invalid file descriptor (not an open file)
[EMFILE]	no more files can be opened; process is at current limit (32).

SEE ALSO

fork(2), open(2)

NAME

execve - replace current process with an executable image from a file

SYNOPSIS

```
#include <gno/gno.h>
int execve(char *pathname, char *cmdline);
```

DESCRIPTION

execve is the preferred method for loading program files to be executed under the GNO system. A new userID is allocated for the process, and the GS/OS System Loader is used to bring the executable file specified by *pathname* into memory. *pathname* can be a partial or complete path. The executable loaded replaces the executable associated with the current process.

If the executable file does not contain an OMF Stack Segment (SEGKIND = \$12), a default stack of 4096 bytes is allocated to the process. The direct-page pointer is set to the bottom of the stack memory (for C programs this is irrelevant).

The parameter *cmdline* is the list of arguments to be passed to the new process (a copy is actually passed). C programs parse *cmdline* automatically, and the individual pieces can be accessed through the *argc/argv* arguments to *main()*. *cmdline* can be accessed from assembly language through the X (high-order word of *cmdline*) and Y (low-order word) registers. However, if the executable file is of file type S16 (\$B3), the *cmdline* argument is ignored and the X&Y registers are set to null (i.e. the command line is only passed to an EXE executable). The 8 characters "BYTEWRKS" are prepended to *cmdline* before being passed to the process (this is the same identifier used by the ORCA shell). This Shell Identifier distinguishes the GNO and ORCA environments from others that don't support the full range of shell calls, and can be accessed from C with the library function *shellid()*. The A register is set to the userID allocated for the process.

GS/OS prefixes 1 and 9 are set to the pathname of the directory containing the executable file; if the length of the path exceeds 64 characters prefix 1 is set to the null prefix (length 0).

The following information is inherited by the new executable: current machine state, controlling TTY, process group ID, and prefixes 0 and 8.

Caught signals are reset to the default action. Ignored signals remain ignored across the **execve**. Any signals in the parent's queue are not passed to the child, and the child is started with no signals blocked. The child inherits all the open files of its parent.

RETURN VALUE

A successful **execve** does not return, as the current executable is replaced with the one specified in the call. If for some reason the call fails, **execve** returns SYSERR (-1), and **errno** is set to one of the following:

[ENOENT]	the <i>pathname</i> specified does not exist
[EIO]	some general I/O error occurred trying to load the executable

BUGS

ORCA/C 1.3 and previous ignore any stack space allocated for it by the GS/OS Loader (which **execve** calls) or by default in **execve**. Stack space in ORCA/C programs is determined by code in the .root object file, and can be set with the `#pragma stacksize` directive. Read the chapter on GNO Compliance in the *GNO Kernel Reference Manual* for more information on this topic. ORCA/C 2.0 and newer use the system-provided stack space.

SEE ALSO

exec(2), **fork(2)**, **wait(2)**, **ioctl(2)**, **tty(4)**, *GNO Kernel Reference Manual*

NAME

fork - start a new process from inside the current application space

SYNOPSIS

```
#include <gno/gno.h>
```

```
int fork(void *addr);
```

DESCRIPTION

fork's argument *addr* is typically the address of a C function, although it can be any valid address inside the IIGS RAM space. **fork** creates a new entry in the process table, and sets up default settings for the new process. The process is allocated 1K (1024 bytes) of stack space, and the direct page is set to the beginning of this memory. The process is executed in 16-bit full native mode, and the registers upon entry to the routine are set as follows:

A	the userID assigned to the process
X	0
Y	0

The child inherits the memory shadowing and machine state parameters of the parent, as well as signal blocking information and the ID of the controlling TTY. In addition, the child inherits all the open files of its parent.

A forked process may share code with other children or the parent. However, this is only allowed in a forward manner; any forked process that exits by function return will be terminated. Note that any shared global variables will need to be moderated with some type of mutual exclusion, either the kernel semaphore(2) routines or custom routines. This includes C stdio routines.

RETURN VALUE

fork returns the process ID of the child, or -1 (SYSERR) if an error occurs, in which case **errno** is set as follows:

[ENOMEM]	not enough memory to create the new process
[EAGAIN]	all process slots full; no more can be created

NOTES

There is no way to pass parameters directly to a child with **fork**(). Use **fork2**() instead.

CAVEATS

Most UNIX **forks** take no parameters; they copy the entire address space of the calling process and return with a different value in the parent and child. Due to hardware limitations, this sort of manipulation isn't possible on the IIGs. UNIX programs utilizing **fork** will have to be modified slightly to work under GNO.

SEE ALSO

fork2(2), **exec**(2), **execve**(2), **wait**(2), **screate**(2)

NAME

`getpid`, `getpgrp`, `getppid` - return information about processes

SYNOPSIS

```
#include <gno/gno.h>
```

```
int getpid(void);  
int getpgrp(int pid);  
int getppid(void);
```

DESCRIPTION

The process ID is a unique value associated with a process, and is needed for many system calls. `pid`'s can range from 0 (Kernel Null Process) to 32767. Some programs use **getpid** to seed random number generators. A much better approach on the IIGS is to use the horizontal and vertical positions of the electron gun, which can be obtained by reading the word value at absolute memory location `$E0C02E`.

getpgrp returns the process group ID of the specified process, *pid*. This function is usually used when you wish to send a signal to all members of a process group using **kill**(2).

RETURN VALUE

getpid: The process ID of the caller is returned. No errors are possible.
getppid: The process ID of the caller's parent is returned. No errors are possible.
getpgrp: The process group ID of the specified process is returned. In the event of an error, **getpgrp** returns -1 and sets **errno** to the appropriate error code:
[ESRCH] *pid* is not a valid process ID

NOTES

getpgrp() is provided solely for compatibility with previous versions of the UNIX operating system. The new Job Control interface should be used exclusively in new software.

SEE ALSO

fork(2), **job control**(2), **ioctl**(2), *GNO Kernel Reference Manual*

NAME

getuid, **geteuid**, **getgid**, **getegid** - get user and group identity
getpgrp - get process group

SYNOPSIS

```
#include <sys/types.h>
uid_t = getuid(void)
uid_t = geteuid(void)
gid_t = getgid(void)
gid_t = getegid(void)
```

DESCRIPTION

Getuid returns the real user ID of the current process, **geteuid** the effective user ID. The real user ID identifies the person who is logged in. The effective user ID gives the process additional permissions during execution of "set-user-ID" mode processes, which use **getuid** to determine the real-user-id of the process that invoked them.

Getgid returns the real group ID of the current process, **getegid** the effective group ID. The real group ID is specified at login time. The effective group ID is more transient, and determines additional access permission during execution of a "set-group-ID" process, and it is for such processes that **getgid** is most useful.

NOTES

The **setr*()** functions are not currently implemented in GNO/ME, because set-uid and set-group-ID behavior is specific to the UNIX filesystem.

SEE ALSO

setreuid(2), **setregid(2)**, **setgid(3)**, **tty(4)**

NAME

ioctl - control device

SYNOPSIS

#include <sys/ioctl.h>

int **ioctl**(int *fd*, unsigned long *request*, void **argp*)

DESCRIPTION

Ioctl performs a variety of functions on open file descriptors. In particular, many operating characteristics of character special files (e.g. terminals) may be controlled with **ioctl** requests. The writeups of various devices in section 4 discuss how **ioctl** applies to them.

An **ioctl request** has encoded in it whether the argument is an "in" parameter or "out" parameter, and the size of the argument *argp* in bytes. Macros and defines used in specifying an **ioctl** request are located in the file <sys/ioctl.h>.

RETURN VALUE

If an error has occurred, a value of -1 is returned and **errno** is set to indicate the error.

ERRORS

Ioctl will fail if one or more of the following are true:

[EBADF] *Fd* is not a valid descriptor.

[ENOTTY] *Fd* is not associated with a character special device.

[ENOTTY] The specified request does not apply to the kind of object that the descriptor *fd* references.

[EINVAL] *Request* or *argp* is not valid.

SEE ALSO

execve(2), **tty**(4)

NAME

tcnewpgrp, **settpgrp**, **tctpggrp** - interface for the new job control model

SYNOPSIS

```
#include <gno/gno.h>
```

```
int tcnewpgrp(int fdtty);  
int settpgrp(int fdtty);  
int tctpggrp(int fdtty, int pid);
```

DESCRIPTION

The job control interface is used to control what processes are ``in the foreground" on a particular terminal. Every tty has a process group. Each process is a member of a process group. A process is a foreground process on a tty if and only if that process and the terminal belong to the same process group. Certain characters typed on a tty with a nonzero process group produce signals sent to every process which is a member of the group (e.g. ^C).

A process is suspended (stopped) if it performs a sufficiently invasive operation on a tty with a different process group. This includes these job control calls, reads from a terminal, and writes to a terminal if configured to do so with **ioctl**(2). When a tty file is first opened, it is assigned process group 0; **init** has process group 0. As **init** launches login processes on various ttys, it assigns process groups to those ttys and processes.

tcnewpgrp(*fdtty*): Allocates a new process group and assigns it to the terminal referred to by *fdtty*. If the calling process is not in the foreground, it is sent SIGTTOU.

settpgrp(*fdtty*): Sets the current process to have the same process group as *fdtty*.

tctpggrp(*fdtty*,*pid*): Sets the tty to the same process group as *pid*, where *pid* is the current process or a descendant of it.

RETURN VALUE

The calls will return 0 if no error occurs; otherwise, they'll return -1 and set **errno** to one of the following:

[EBADF]	<i>fdtty</i> is not a valid file descriptor
[ENOTTY]	<i>fdtty</i> does not refer to a terminal file
[ESRCH]	<i>pid</i> is not a valid process identifier

NOTES

Following are some example uses of the job control interface.

- Forking a pipeline in a job-control shell: The shell starts with **tcnewpgrp**(*fdtty*), so that the tty is in the new process group before there are even any children. It then forks each process in the pipeline. Each process does **settpgrp**(*fdtty*), thus joining the new process group.
- Handling a stopped child process: When the shell sees that a pipeline has stopped or exited, it does **tctpggrp**(*fdtty*,**getpid**()) to set the tty to its own process group. To resume the pipeline it does **tctpggrp**(*fdtty*,*pid*) where *pid* is one of the child processes, then sends SIGCONT.

- Starting a process under a new tty: When, for instance, telnetd wants to grab a pseudo-tty, it opens the pty and forks a child process. The child does **tcnewpgrp**(fdtty) to give the tty a real process group, then **settpgrp**(fdtty) to place itself into the foreground.

Security under this scheme is trivial. There is no way a process can join a process group except by **settpgrp**(), and that requires a descriptor open to a tty with that pgrp. To make a tty have that pgrp requires either **tcnewpgrp**(), in which case nobody else is using the pgrp, or **tctpgrp**(), which reduces to the first problem of having a process in the process group. End of security proof. (Wasn't that easy?) Note that 'using' must be defined as use both by ttys and by processes; the system keeps a table of pgrps, each with a total tty + process reference count. When the reference count reaches zero, the pgrp is automatically deallocated.

SEE ALSO

ioctl(2), **tty**(4), **signal**(2), **kill**(2), *GNO Shell Reference Manual*

CREDITS

This job control interface was designed by Dan Bernstein (brnstnd@kramden.acf.nyu.edu). He was inspired by Chris Torek, and dedicated the system to Mark Teitelbaum. The text of this manpage is derived from his original specifications.

The GNO/ME implementation was written strictly from specs.

NAME

kill - send a signal to a process

SYNOPSIS

```
#include <gno/gno.h>
```

```
int kill(int pid, int sig)
```

DESCRIPTION

kill is used to send a signal to a process or group of processes. Signals are software interrupts; they act just like hardware interrupts and can also be used for basic IPC (Inter-process communication). The various signals are described in **signal(2)**.

sig can be a signal number, or it can be 0, in which case no signal is sent, but error checking is done (this can be used to verify a process ID). If *sig* has been blocked (**sigblock(2)**), the signal is deferred until it is unblocked, and **kill** returns immediately. Any previously pending signals of the same *sig* are lost (i.e. signals are not stacked).

If *pid* is 0, the signal is sent to all processes with the same process group ID as the caller, except for system processes.

Processes may signal themselves, in which case the signal handler is invoked immediately (if one is installed).

RETURN VALUE

Normally **kill** returns SYSOK (0). The following conditions can cause **kill** to return SYSERR (-1):

[ESHRC]	<i>pid</i> does not correspond to an existing process
[EINVAL]	<i>sig</i> specifies an invalid signal number

SEE ALSO

sigblock(2), **signal(2)**, **wait(2)**

BUGS

Do not attempt to send signals from inside a CDA (Classic Desk Accessory) or interrupt handler.

NAME

pipe - create an interprocess communication channel

SYNOPSIS

```
#include <gno/gno.h>
int pipe(int fildes[2]);
```

DESCRIPTION

The **pipe** system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor *fildes[1]* up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor *fildes[0]* will pick up the data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent fork calls) will pass data through the pipe with Read and Write calls.

The shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) returns an end-of-file.

A signal (SIGPIPE) is generated if a write on a pipe with only one end is attempted.

RETURN VALUE

The function value zero is returned if the pipe was created; -1 if an error occurred.

ERRORS

The pipe call will fail if:

- | | |
|----------|---|
| [EMFILE] | Too many descriptors are active. |
| [ENFILE] | The system file table is full. |
| [EFAULT] | The fildes buffer is in an invalid area of the process's address space. |

SEE ALSO

GNO Shell Users Manual, **read(2)**, **write(2)**, **fork(2)**

BUGS

Should more than 4096 bytes be necessary in any pipe among a loop of processes, deadlock will occur.

NOTES

In the above text, mention is made to UNIX read and write calls. On the Apple IIgs, these refer to any system calls that do I/O, namely GS/OS ReadGS, WriteGS; TextTools calls; and C library I/O calls such as read, fread, etc.

NAME

screate, sdelete, swait, ssignal - semaphore operations

SYNOPSIS

```
#include <gno/gno.h>
```

```
int swait(int sem);  
int ssignal(int sem);  
int screate(int count);  
int sdelete(int sem);
```

DESCRIPTION

screate is used to allocate a semaphore from the kernel semaphore manager. Semaphores are the most basic form of interprocess communication, and these routines provide the power necessary to solve a large number of synchronization and communication problems. (See an Operating Systems text).

The initial *count* determines how many times **swait** can be called before processes are blocked. *count* must be ≥ 0 , and is usually set to 1. **screate** returns a semaphore ID number as an integer. This ID must be used in all the other semaphore calls.

sdelete releases the specified semaphore, and returns all processes that were blocked to a ready state.

swait decrements the value of the semaphore (initially specified by *count*) by 1. If the semaphore count is less than zero, the process is blocked and queued for release by **ssignal**.

ssignal increments the semaphore count by one. If the semaphore count is less than zero, **ssignal** releases arbitrarily a process that had been blocked; FIFO operation is not guaranteed.

RETURN VALUE

All the functions return SYSERR (-1) if an error occurs, and an OK (0) if no error occurs.

BUGS

There is currently no mechanism for deallocating semaphores that are orphaned by abnormal process termination.

HISTORY

These semaphore routines were designed for XINU, written by Douglas Comer.

NAME

setdebug - set debugging output options

SYNOPSIS

```
#include <gno/gno.h>
```

```
int setdebug(int options);
```

DESCRIPTION

setdebug enables and disables various debugging routines built into the kernel. The routines display useful debugging information to stderr (except for `dbgSIG`, see BUGS). Debug output is enabled by setting the corresponding bit in options, according to the following table. To turn off all debugging output, options should be set to 0. The various debug options are #defined in `<gno/gno.h>`.

dbgGSOS	prints out the call numbers of any GS/OS or ORCA/shell calls that are made. The number is printed in hexadecimal format and is prefixed with a '\$' character. For this and the other GS/OS call debug options, the entire call sequence is enclosed in parenthesis '()' to ease tracing multiple levels of calls.
dbgPATH	If this flag is set, every time a filename argument to a GS/OS or shell call is fully expanded the expanded version is displayed as follows: "EP: <fullpath>".
dbgERROR	For every GS/OS call that is made, if an error occurs the error code is printed in inverse lettering in hexadecimal format. The code is prefixed with a '#' to distinguish the error code from a call code on terminals that do not support inverse mode. If no error occurs on the call, no code is printed. This option has no effect unless <code>dbgGSOS</code> is also enabled.
dbgSIG	This flag enables signal tracing. Each time a signal is sent, whether by <code>kill(2)</code> , job control or keyboard, the signal number and target process is displayed. The format is: "kill (-signum): pid: tpid".
dbgSYSCALL	The parameter lists to common system calls are displayed by this option flag. The actual format of the output varies from call to call. The calls that currently support this flag are execve(2) , fork(2) , and the job control routines.
dbgPBLOCK	The memory address of GS/OS and Shell parameter blocks is printed for each call. As with <code>dbgERROR</code> , this option has no effect unless <code>dbgGSOS</code> is also enabled.

RETURN VALUE

setdebug returns the previous value of the debug options word.

SEE ALSO

fork(2), **execve(2)**, **ioctl(2)**, **kill(2)**

BUGS

Due to problems associated with signals that are sent during process termination, `dbgSIG` prints its information to standard output instead of standard error.

NAME

sigblock, **sigmask** - temporarily block signals

SYNOPSIS

```
#include <signal.h>
```

```
long sigblock(long mask);  
#define sigmask(signum)
```

DESCRIPTION

sigblock is used to temporarily block the reception of signals. The input parameter *mask* is a bit vector that specifies which signals are to be blocked; a 1 in a bit *n* will block signal *n*+1. The mask is bitwise-ored with the current signal mask to create the new signal mask.

sigmask is a macro that can be used to calculate signal masks for **sigblock**. It takes a signal number (*signum*) as an argument and returns a mask that can then be passed to **sigblock**.

If a signal is sent to a process but is blocked, the event is recorded for later release by **sigsetmask**(2). Blocked signals are not stacked; further occurrences of a blocked signal will overwrite any previous pending signal of the same *signum*.

It is not possible to block SIGKILL, SIGCONT, or SIGSTOP. This restriction is silently enforced by the system.

RETURN VALUE

The previous value of the signal mask is returned.

SEE ALSO

kill(2), **sigsetmask**(2), **signal**(2)

NAME

signal - a simplified software signal interface

SYNOPSIS

```
#include <signal.h>
```

```
void (*signal)(int sig; void (*func(void))(void))
```

DESCRIPTION

Signals are a basic form of IPC (inter-process communication), and are generally used to notify a process of some atypical event (although there is little restriction on actual use). For example, signals are sent in each of the following situations: user typing certain chars at a terminal (^C, ^Z, etc.); execution of an invalid instruction; by request of another process (**kill**); stack overflow; a process making an input request while running in the background; an attempt to write to a pipe with no reader.

Most signals cause termination, unless a handler is installed, or the signal is set to be ignored. Certain signals cannot have their default action modified; the system silently enforces this restriction. The following is a list of signals and default actions (taken from signal.h).

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3	quit
SIGILL	4	illegal instruction
SIGTRAP	5	trace trap
SIGABRT	6	abort (generated by abort(3) routine)
SIGEMT	7	emulator trap
SIGFPE	8	arithmetic exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10	bus error
SIGSEGV	11	segmentation violation
SIGSYS	12	bad argument to system call
SIGPIPE	13	write on a pipe or other socket with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGURG	16@	urgent condition present on socket
SIGSTOP	17 +	stop (cannot be caught, blocked, or ignored)
SIGTSTP	18 +	stop signal generated from keyboard
SIGCONT	19@	continue after stop (cannot be blocked)
SIGCHLD	20@	child status has changed
SIGCLD	20	System V name for SIGCHLD
SIGTTIN	21 +	background read attempted from control terminal
SIGTTOU	22 +	background write attempted to control terminal
SIGIO	23@	input/output possible on a file descriptor
SIGPOLL	SIGIO	System V name for SIGIO
SIGXCPU	24	exceeded CPU time limit
SIGUSR1	30	user defined signal 1
SIGUSR2	31	user defined signal 2

If *func* is SIG_DFL, the default action for the signal is reinstalled. This is normally termination if the signal isn't ignored or caught. Signals marked with an @ are discarded, and signals marked with | + cause the process to stop. If *func* is SIG_IGN, any future occurrences of the signal are discarded, as well as any pending instances. Any other value is treated as the address of a handler

routine. The system will block further occurrences of the signal before the handler is called, and will unblock the signal automatically upon return from the handler. The handler remains installed after return, unlike earlier UNIX signal facilities.

If a signal occurs during certain system calls (**wait()**, and input from a TTY), the call is automatically restarted upon return from the handler.

A forked child inherits all signal information, including pending signals and blocking and handler information. **exec()** and **execve()** restore all signal information to defaults and purge pending signals.

NOTES

The signal handler should be defined as follows:

void handler (int *sig*, int *code*)

sig is the signal that invoked the handler, and *code* is additional information about the interrupt condition. Currently, *code* is always 0. The handler should probably also be compiled using the `#pragma databank 1` directive, in the event the signal handler is not in the same bank as the C global data segment (the handler is called with the data bank equal to the program bank).

RETURN VALUE

The previous action is returned on a successful call. Otherwise, -1 is returned. [EINVAL] will occur on any of the following conditions:

sig specifies an invalid signal number.

An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

An attempt is made to ignore SIGCONT (by default SIGCONT is ignored).

CAVEATS

ORCA/C already provides a **signal()** function, but it doesn't do a whole lot. GNO's **signal.h** file replaces the one that comes with ORCA.

SEE ALSO

execve(2), **fork(2)**, **kill(2)**, **sigblock(2)**, **sigsetmask(2)**, **wait(2)**, **tty(4)**

NAME

sigpause - suspend process until a signal arrives

SYNOPSIS

```
#include <signal.h>
int sigpause(long int mask);
```

DESCRIPTION

sigpause suspends execution of the calling process until a signal arrives. The *mask* parameter is assigned to the set of blocked signals (**sigsetmask**), and then the process is paused. When a signal arrives, the regular signal handler (if any) is executed, and then the original signal mask is restored before returning to the caller. Usually *mask* is 0 (zero) to pause until any signal arrives.

sigpause is normally used in situations where one must protect a critical section. A typical use begins with **sigblock** to block a signal (and enable mutual exclusion); variables modified on the occurrence of that signal are then manipulated, code is executed, etc. To end the critical section and wait for more work, **sigpuase** is called.

RETURN VALUE

sigpause always returns an error (-1) and sets **errno** to **EINTR**.

SEE ALSO

signal(2), **sigblock(2)**, **sigsetmask(2)**

NAME

sigsetmask - set signal mask

SYNOPSIS

```
#include <signal.h>
```

```
long sigsetmask(long mask);
```

```
#define sigmask(signum)
```

DESCRIPTION

sigsetmask is usually used to restore signal masks after modification by **sigblock**. The parameter *mask* is the absolute value the process signal mask will be set to (compare to **sigblock**, which adds the argument to the set of blocked signals).

If there are pending instances of signals which become unblocked by the **sigsetmask** call, they are 'released' into the system signal queue and their 'pending' status is cleared. The system signal queue is maintained by the kernel null process, and is used in situations where signals could not normally be sent (such as interrupt handlers).

sigmask is a macro that can be used to calculate signal masks for **sigsetmask**. It takes a signal number, as listed in **signal(2)**, as an argument and returns a mask corresponding to that signal.

RETURN VALUE

The previous value of the signal mask is returned.

CAVEATS

If somehow the process re-blocks a signal released by **sigsetmask** before the system signal queue processes it, it will be blocked and marked as pending. This can happen if a signal handler makes a **sigblock** call.

SEE ALSO

kill(2), **sigblock(2)**, **signal(2)**

NAME

stat, fstat, lstat - return status information on indicated files

SYNOPSIS

```
#include <sys/stat.h>
```

```
int stat(const char *filename, struct stat *s_buf);
int fstat(int filedес, struct stat *s_buf);
int lstat(const char *filename, struct stat *s_buf);
```

DESCRIPTION

These calls are used to retrieve status information about files. They do much the same thing as the GS/OS call GetFileInfo, except that they return the information in a format compatible with UNIX's stat calls, and also provide information about pipes and GNO Character Devices.

stat takes arguments *filename*, a NUL-terminated string naming the file to get information on, and *s_buf*, a pointer to a **stat** structure, defined in `<sys/stat.h>`. *filename* can be a partial or a complete path. The miscellaneous types in **struct stat**, below, are defined in `<sys/types.h>`, automatically included by `stat.h`.

```
struct stat {
    dev_t    st_dev;        /* ID number of device file resides on */
    ino_t    st_ino;        /* inode number of file */
    unsigned short st_mode; /* type of file and mode */
    short    st_nlink;      /* number of links to file = 0 */
    uid_t    st_uid;        /* user id = 0 */
    gid_t    st_gid;        /* group id = 0 */
    dev_t    st_rdev;       /* device type ID */
    off_t    st_size;       /* length of file in bytes */
    time_t    st_atime;     /* last access time (same as mod time on Apple
IIGS) */
    int      st_spare1;     /* reserved */
    time_t    st_mtime;     /* last modification time */
    int      st_spare2;     /* reserved */
    time_t    st_ctime;     /* file creation time */
    int      st_spare3;     /* reserved */
    long     st_blksize;    /* size in bytes of blocks on filesystem */
    long     st_blocks;     /* number of blocks file uses */
    long     st_spare4[2];  /* reserved */
};
```

The items marked 'reserved' are not currently used but are reserved for future expansion; do not use these fields for any reason. `st_dev` is the device number the file resides on. This number is the same as the GS/OS device ID number. `st_rdev` is not currently used, but may in the future designate a device type code.

`st_mode` is a bit field representing access mode and type of the file. The flags in `st_mode` are as follows:

```
#define S_IFDIR    0040000 /* directory */
#define S_IFCHR    0020000 /* character special */
#define S_IFBLK    0060000 /* block special */
#define S_IFREG    0100000 /* regular */
#define S_IFLNK    0120000 /* symbolic link */
```



```
#define S_IFSOCK 0140000 /* socket or pipe */
#define S_IRREAD 0000400 /* read permission, owner */
#define S_IWRITE 0000200 /* write permission, owner */
#define S_IEXEC 0000100 /* file is an executable, owner */
```

fstat is similar to **stat** except the argument is an open file descriptor *filedes*. If *filedes* refers to a character device or pipe, the entire *s_buf* is set to 0 and only *st_mode* and *st_dev* are set.

lstat is similar to **stat**, but if the *filename* is a link then information is returned about the link file instead of the file linked to.

RETURN VALUE

If the call completes without mishap, **stat** returns a 0. If an error occurs, **stat** returns -1 and sets *errno* to one of the following:

[ENOENT] *filename* does not specify an existing file or directory
[ENOTDIR] an element of *filename* is not an expected subdirectory

fstat can additionally return

[EBADF] *filedes* does not refer to an open file descriptor

BUGS

GNO does not yet support hard or symbolic file links on the IIGS. Therefore, **lstat** operates exactly like **stat**. But if there's a case where **lstat** might be appropriate at a time when links are supported, then use **lstat** instead and be ready for the future.

fstat doesn't do anything clever with all the unused fields in `struct stat` when its argument is a pipe or terminal.

NAME

statfs - get file system statistics

SYNOPSIS

```
#include <sys/vfs.h>
```

```
int statfs(char *path, struct statfs *buf)
int fstatfs(int fd, struct statfs *buf)
```

DESCRIPTION

statfs returns information about a mounted file system. *path* is the path name of any file within the mounted filesystem. *Buf* is a pointer to a **statfs** structure defined as follows:

```
typedef struct {
    long val[2];
} fsid_t;

struct statfs {
    long f_type;      /* type of info, zero for now */
    long f_bsize;     /* fundamental file system block size */
    long f_blocks;    /* total blocks in file system */
    long f_bfree;     /* free blocks */
    long f_bavail;    /* free blocks available to non-superuser */
    long f_files;     /* total file nodes in file system */
    long f_ffree;     /* free file nodes in fs */
    fsid_t f_fsid;    /* file system id */
    long f_spare[7]; /* spare for later */
};
```

Fields that are undefined for a particular file system are set to -1. **fstatfs** returns the same information about an open file referenced by descriptor *fd*.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, -1 is returned and the global variable **errno** is set to indicate the error.

ERRORS

statfs fails if one or more of the following are true:

ENOTDIR	A component of the path prefix of <i>path</i> is not a directory.
EINVAL	<i>path</i> contains a character with the high-order bit set.
ENAMETOOLONG	The length of a component of <i>path</i> exceeds 255 characters, or the length of <i>path</i> exceeds 1023 characters.
ENOENT	The file referred to by <i>path</i> does not exist.
EACCES	Search permission is denied for a component of the path prefix of <i>path</i> .
EIO	An I/O error occurred while reading from or writing to the file system.

fstatfs fails if one or both of the following are true:

EBADF	<i>fd</i> is not a valid open file descriptor.
EIO	An I/O error occurred while reading from or writing to the file system.

NAME

truncate, ftruncate - set a file to a specified length

SYNOPSIS

```
#include <sys/types.h>
```

```
int truncate(char *path, off_t length)
int ftruncate(int fd, off_t length)
```

DESCRIPTION

truncate() causes the file referred to by *path* (or for **ftruncate()** the object referred to by *fd*) to have a size equal to *length* bytes. If the file was previously longer than *length*, the extra bytes are removed from the file. If it was shorter, bytes between the old and new lengths are read as zeroes. With **ftruncate()**, the file must be open for writing.

RETURN VALUES

truncate() returns:

0 on success.

-1 on failure and sets **errno** to indicate the error.

ERRORS

truncate() may set **errno** to:

EACCES Search permission is denied for a component of the path prefix of *path*.
Write permission is denied for the file referred to by *path*.

EIO An I/O error occurred while reading from or writing to the file system.

EISDIR The file referred to by *path* is a directory.

ENAMETOOLONG The length of the *path* argument exceeds {**PATH_MAX**}.
A pathname component is longer than {**NAME_MAX**} (see **sysconf** (2V)) while {**_POSIX_NO_TRUNC**} is in effect (see **pathconf** (2V)).

ENOENT The file referred to by *path* does not exist.

ENOTDIR A component of the path prefix of *path* is not a directory.

EROFS The file referred to by *path* resides on a read-only file system.

ftruncate() may set **errno** to:

EINVAL *fd* is not a valid descriptor of a file open for writing.
fd refers to a socket, not to a file.

EIO An I/O error occurred while reading from or writing to the file system.

SEE ALSO

open (2)

BUGS

These calls should be generalized to allow ranges of bytes in a file to be discarded.

NAME

wait, **WIFSTOPPED**, **WIFSIGNALED**, **WIFEXITED** - wait for process to terminate or stop

SYNOPSIS

```
#include <sys/wait.h>
```

```
int wait(union wait *statusp);
WIFSTOPPED(union wait status);
WIFSIGNALED(union wait status);
WIFEXITED(union wait status);
```

DESCRIPTION

wait blocks the caller until a signal is received or one of its child processes terminates. If any child has died and this has not been reported using **wait**, return is immediate, returning the process ID and exit status of one of those children. If that child had died, it is discarded. If there are no children, return is immediate with the value -1 returned. If there are processes that have not been reported by **wait**, the caller is blocked.

If *status* is not a NULL pointer, then on return from a successful **wait** call the status of the child process whose process ID is the return value of **wait** is stored in the **wait** union pointed to by *status*. The *wstatus* member of that union is an int; it indicates the cause of termination and other information about the terminated process in the following manner:

- If the low-order 8 bits of *wstatus* are equal to 0177 (hex 0xFF), the child process has stopped; the high-order 8 bits of *wstatus* contain the number of the signal that caused the process to stop. See **signal(2)**.
- If the low-order 8 bits of *wstatus* are non-zero and are not equal to 0177, the child process terminated due to a signal; the low-order 7 bits of *wstatus* contain the number of the signal that terminated the process.
- Otherwise, the child process terminated due to an **exit()** call; the high-order 8 bits of *wstatus* contain the low-order 8 bits of the argument that the child process passed to **exit** or GS/OS Quit.

Other members of the **wait** union can be used to extract this information more conveniently:

- If the *wstopval* member has the value **WSTOPPED**, the child process has stopped; the value of the *wstopsig* member is the signal that stopped the process.
- If the *wtermsig* member is non-zero, the child process terminated due to a signal; the value of the *wtermsig* member is the number of the signal that terminated the process.
- Otherwise, the child process terminated due to an **exit()** call; the value of the *wretcode* member is the low-order 8 bits of the argument that the child process passed to **exit()**.

The other members of the **wait** union merely provide an alternate way of analyzing the status. The value stored in the *wstatus* field is compatible with the values stored by versions of the UNIX system, and an argument of type `int *` may be provided instead of an argument of type `union wait *` for compatibility with those versions.

WIFSTOPPED, **WIFSIGNALED**, **WIFEXITED**, are macros that take an argument *status*, of type `'union wait'`, as returned by **wait()**. **WIFSTOPPED** evaluates to true (1) when the

process for which the **wait** call was made is stopped, or to false (0) otherwise. **WIFSIGNALED** evaluates to true when the process was terminated with a signal. **WIFEXITED** evaluates to true when the process exited by using an **exit(2)** call.

If **wait** returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

ERRORS

wait will fail and return immediately if one or more of the following are true:

- | | |
|----------|---|
| [ECHILD] | The calling process has no existing unwaited-for child processes. |
| [EFAULT] | The status or rusage arguments point to an illegal address. |
| [EINTR] | The wait call was interrupted by a caught signal. |

SEE ALSO

signal(2), **exit(3)**, **rexit(3)**, **execve(2)**

NOTES

If a parent process terminates without waiting on its children, the Kernel Null Process (process ID = 0) inherits the children.

wait is automatically restarted when a process receives a signal while awaiting termination of a child process, if the signal is not caught; i.e. **signal()** handler value is **SIG_DFL** or **SIG_IGN**.

GSString255Ptr **__C2GSMALLOC**(char *s)

Converts a C-style string to a Class 1 GS/OS string, allocating space for the GS/OS string from C's malloc() routine. You must specifically deallocate the string with free() when you're through with it.

char ***__GS2CMALLOC**(GSString255Ptr g)

Converts a Class 1 GS/OS string to a C-style string, allocating space for the C string from C's malloc() routine. You must specifically deallocate the string with free() when you're through with it.

char ***__GS2C**(char *s, GSString255Ptr g)

Converts a Class 1 GS/OS string to a C string; the buffer space for the C string must be allocated beforehand by the caller. The return value is the address of the C string passed as argument s.

int **_mapErr**(int err)

Tries to map a GS/OS error code (err) to a UNIX errno code (return value). If there is no direct mapping, EIO is returned.

access

```
#include <unistd.h>
```

```
int access(char *name, int mode)
```

Returns TRUE (1) if the file specified by *name* can be accessed according to *mode* by the calling process. Values of mode are declared in <unistd.h> and are as follows:

F_OK - returns true if the file exists

X_OK - returns true if the process has execution permissions for the file

W_OK - returns true if the process has write permissions for the file

R_OK - returns true if the process has read permissions for the file

bcopy bzero

```
#include <string.h>
```

```
void bcopy(char *b1, char *b2, size_t n)
```

```
void bzero(char *buf, size_t n)
```

bcopy() copies *n* bytes from memory address *b1* to memory address *b2*. **bcopy()** is functionally similar to **memcpy()**, except that **bcopy** copies from the first argument to the second argument, whereas **memcpy()** copies from the second argument to the first argument. If the memory areas overlap, the results are unpredictable. **bcopy()** is provided for compatibility with BSD source code.

bzero() clears *n* bytes of memory starting at *buf* to 0 (zero). This call is functionally equivalent to **memset(buf,0,n)** and is included for BSD source code compatibility.

See Also: **memcpy**, **memset**, *ORCA/C 2.0 Manual*

chdir

```
#include <unistd.h>
```

```
int chdir(const char *pathname)
```

Changes the current working directory (GS/OS prefix 0) to the pathname specified by *pathname*. If an error occurs changing the prefix, -1 is returned and the error code is placed in **errno**.

crypt

```
char *crypt(char *pw, char *salt)
```

crypt is used to encrypt passwords for storage in the `/etc/passwd` file, and also to validate passwords entered in the `login` and `passwd` programs. *pw* is the password to encrypt, a NUL-terminated string. *salt* is a two-character encryption key that should be randomly generated by the caller in the case of encrypting a new password, or should be taken as the first two characters of the `/etc/passwd` password entry in the case of validating a password.

crypt returns a pointer to the encrypted password, which is formatted as printable ASCII characters and is NUL terminated. A static buffer is used to hold the result, so to be sure the encrypted password is not overwritten by a subsequent call to **crypt** copy it before use.

See also: **getpass**, **getpwent**

errno strerror perror

```
char *strerror(int errnum)
void perror(char *s)
extern int errno;
```

These routines are as documented in the ORCA/C manual, except that they support the full range of GNO's `errno` codes. **errno** is the variable that most library and kernel calls place their return status in. The codes are defined symbolically in `<errno.h>` and are listed here:

EDOM	domain error
ERANGE	number too large, too small, or illegal
ENOMEM	Not enough memory
ENOENT	No such file or directory
EIO	I/O error
EINVAL	Invalid argument
EBADF	bad file descriptor
EMFILE	too many files are open
EACCESS	access bits prevent the operation
EEXIST	the file exists
ENOSPC	the file is too large
EPERM	Not owner
ESRCH	No such process
EINTR	Interrupted system call
E2BIG	Arg list too long
ENOEXEC	Exec format error
ECHILD	No children
EAGAIN	No more processes
ENOTDIR	Not a directory
ENOTTY	Not a terminal
EPIPE	Broken pipe
ESPIPE	Illegal seek
ENOTBLK	not a block device
EISDIR	not a plain file

fsync

```
int fsync(int fd)
```

Causes the operating system to flush any I/O buffers associated with the file referenced by file descriptor *fd* to disk. This ensures that all information is up to date, in the event of a system crash. This call is only needed in special circumstances, as when several daemon processes are all modifying the same file simultaneously (currently impossible with existing IIGS filesystems). This call is basically a **FlushGS**.

ftruncate

```
int ftruncate(int fd, off_t length)
```

Causes the EOF marker for the file specified by file descriptor *fd* to be set to *length*. In the event of an error, *ftruncate* returns -1 and sets *errno*.

getgrnam getgrgid getgrent setgrent setgroupent endgrent

```
#include <sys/types.h>
#include <grp.h>
struct group *getgrnam(const char *name);
struct group *getgrgid(gid_t gid);
struct group *getgrent(void);
int setgrent(void);
int setgroupent(int stayopen);
void endgrent(void);
(POSIX)
```

This family of functions should be used to access the groups database; applications should never read */etc/groups* directly, as the implementation of the groups database is subject to change.

getgrnam() reads the group database based on group name. It looks up the supplied group name and returns a pointer to a struct group (see below), or NULL on an error.

getgrgid() is similar to **getgrnam()**, except that instead of looking up group information based on group name, a group ID is passed.

To scan the groups database linearly, start the scan with either **setgrent()** or **setgroupent()**. The two functions are identical for pure scanning operations, but have different behavior when mixing scan calls with **getgrnam()** or **getgrgid()**.

After calling **setgrent** or **setgroupent**, the scan is set to the first entry in the database. **getgrent** returns a pointer to the current entry and moves the scan to the next entry. In the event of an error, **getgrent** returns NULL. When the program is done with the database, **endgrent** should be called.

If **getgrnam** or **getgrgid** is called while scanning the group database, the database will be closed unless it was opened by calling **setgroupent** with an argument of 1. This indicates "keep open" mode, and allows fast random access of the database with the **getgrnam** and **getgrgid** functions (which would otherwise open and close the database for every call).

getopt getopt_restart

```
#include <getopt.h>
int getopt(int argc, char * const *argv, const char *optstring);
int getopt_restart(void)
```



```
extern char *optarg;
extern int optind;
```

Getopt helps parse command line options as are often used by UNIX utilities. It handles simple flags (such as "ls -l") and also flags with arguments ("cc -o prog prog.c").

Getopt returns the next option letter in argv that matches a letter in *optstring*. *Optstring* is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. **Optarg** is set to point to the start of the option argument on return from **getopt**.

Getopt places in **optind** the argv index of the next argument to be processed. Because **optind** is external, it is normally initialized to zero automatically before the first call to **getopt**.

When all options have been processed (i.e., up to the first non-option argument), **getopt** returns EOF. The special option -- may be used to delimit the end of the options; EOF will be returned, and -- will be skipped.

Getopt prints an error message on stderr and returns a question mark (?) when it encounters an option letter not included in *optstring*.

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options a and b, and the options f and o, both of which require arguments:

```
main(int argc, char **argv)
{
    int c;
    extern int optind;
    extern char *optarg;

    while ((c = getopt(argc, argv, "abf:o:")) != EOF)
        switch (c) {
            case 'a':
                if (bflg)
                    errflg++;
                else
                    aflg++;
                break;
            case 'b':
                if (aflg)
                    errflg++;
                else
                    bproc();
                break;
            case 'f':
                ifile = optarg;
                break;
            case 'o':
                ofile = optarg;
                break;
            case '?':
            default:
                errflg++;
                break;
        }
    if (errflg) {
        fprintf(stderr, "Usage: ...");
    }
}
```



```

        exit(2);
    }
    for (; optind < argc; optind++) {
        .
    }
}

```

It is not obvious how '-' standing alone should be treated; this version treats it as a non-option argument, which is not always right. Option arguments are allowed to begin with '-'; this is reasonable but reduces the amount of error checking possible.

getopt_restart should be used in restartable programs, before the first call to **getopt**, to reinitialize the **optind** and **optarg** variables.

getpass

```
char *getpass(const char *prompt)
BSD
```

Prompts the user for a password, and returns a pointer to a NUL-terminated string which contains the password the user typed. A password may be up to 8 characters long, and if the string the user types is longer than that the returned string is truncated to 8 characters. Argument *prompt* is the string to print before requesting input. Input characters are obscured - that is, not echoed - as the user types them. The backspace and delete keys may be used to edit input, although in practice this is difficult to use because the user cannot see what he types.

A static buffer is used to hold the password, so to be sure the password is not overwritten by a subsequent call to **getpass**, copy it before use.

See also: **crypt**, **getpwent**

getpwnam getpwuid endpwent setpwent

```
#include <pwd.h>
struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);
void endpwent(void);
struct passwd *getpwent(void);
int setpwent(void);
```

The family of functions defined in `<pwd.h>` are used for accessing the `/etc/passwd` user database. Programs should never access this database directly, as the file format or other implementation details may change in the future.

getpwnam() reads the user database based on user name. The argument *name* is a pointer to the user name to lookup. **getpwnam()** returns a pointer to a `passwd` structure or `NULL` on an error.

getpwuid() reads the user database based on a user ID code. Argument *uid* is the user ID to return information on. **getpwuid()** returns a pointer to a `passwd` structure or `NULL` on an error.

The remaining three functions are used for scanning the user database. The database is initialized by using the **setpwent()** function; an internal access marker is set to the first entry in the database.

getpwent() is used to retrieve the current entry, returning a pointer to a `passwd` structure, and moving the marker to the next entry. If there are no more entries to scan, **getpwent()** returns a `NULL` pointer. If the

database should be scanned again, **setpwent()** may be called again to reset the marker to the first entry. In the event of an error accessing the database, NULL is returned.

When the application is through with the database, it should call **endpwent()**.

```
struct    passwd { /* see getpwent(3) */
    char   *pw_name;      /* pointer to user name */
    char   *pw_passwd;    /* pointer to encrypted password */
    int    pw_uid;        /* user ID */
    int    pw_gid;        /* group ID */
    int    pw_quota;      /* 'quota' field - not used */
    char   *pw_comment;   /* pointer Comment field */
    char   *pw_gecos;      /* not used */
    char   *pw_dir;       /* pointer to user's '$home' directory name */
    char   *pw_shell;     /* pointer to path of user's login shell */
};
```

Not all of the string entries in struct passwd are used in GNO/ME, but those that are are all NUL-terminated strings.

getwd

```
#include <unistd.h>
char *getwd(char *pathname)
```

Gets the current working directory (GS/OS prefix 0) and copies it to the string space pointed to by *pathname*. *pathname* must point to a buffer large enough to hold the largest conceivable pathname. In practice, a 256 byte buffer works well, but with the plethora of GS/OS file systems now available 256 may be much too small. Due to this problem, we recommend you use **getwd** carefully, and with a future GNO release switch to **getcwd** (not yet available).

If an error occurs during the operation, **getwd** returns NULL and places the error code in **errno**. Otherwise, **getwd** returns the prefix in *pathname*.

gtty stty

```
#include <sgtty.h>
int gtty(int filedес, struct sgttyb *argp)
int stty(int filedес, struct sgttyb *argp)
```

Set and get TTY status information in the sgttyb structures pointed to by the argument argp. See **ioctl(2)** and **tty(4)** for more details. These routines are basically short-cuts to **ioctl(filedes, TIOCSETP, &structure)** and **ioctl(filedes, TIOCGETP, &structure)**.

index rindex

```
char *index(char *a, int b)
char *rindex(char *a, int b)
(BSD)
```

These functions are identical to **strchr()** and **strrchr()**, respectively. See your C compiler manual for more information. These functions are provided only for compatibility with BSD source code.

isatty

```
#include <sgtty.h>
int isatty(int filedes)
```

This function returns true (1) if the file descriptor refers to a TTY (this includes PTYs) file. For all other types of descriptors, false (0) is returned.

login

```
#include <utmp.h>
void login(struct utmp *ut)
```

Writes the /etc/utmp structure pointed to by *ut* to the utmp file. The slot in /etc/utmp actually written to depends on the return value of the **ttyslot()** function, which maps each tty device to a unique slot number, based on the contents of /etc/ttys.

This function should not generally be used by application code.

mkdir

```
int mkdir(char *dirname)
```

Creates a subdirectory (folder) with the name specified by *dirname*. Similar to the shell '**mkdir**' command.

mktemp mkstemp

```
char *mktemp(char *path)
int mkstemp(char *path)
```

Creates a filename based on the string *path* that is guaranteed to be unique. The string *path* must have the following format:

"/volume/dir1/.../dirX/fileXXXXXX" (Colons are also accepted as delimiters)

The 'XXXXXX' at the end of *path* is filler space that will be replaced with a string that will make *path* a unique filename.

The unique string is generated by using the current process ID and a single character ASCII value; this may change in the future, and as such this behavior should not be relied upon.

mktemp() does not actually create any files, as compared with **tmpfile()** in the C library.

mkstemp() does create a file by calling **open()** on a unique pathname generated with **mktemp()**.

mktemp() returns a pointer to the new pathname (*path*), and **mkstemp()** returns a file descriptor to the new file, as would be returned by **open()**.

open creat close read write lseek

```
#include <fcntl.h>
int creat(const char *path, int mode)
int open(const char *path, int oflag, ...)
int close(int fileds)
```



```
int read(int filds, void *buf, size_t count)
int write(int filds, void *buf, size_t count)
long lseek(int filds, long offset, int whence)
```

These are similar to the low-level I/O routines provided by ORCA/C. However, the GNO versions of these routines deal with actual GS/OS refNums for filds. (ORCA/C's versions use a special library-maintained definition of file descriptor in order to fake the UNIX **dup()** system call. Here they revert to standard UNIX usage because GNO provides a real **dup(2)** handled within the kernel).

open() uses **vararg** (variable argument) parameters. The third parameter is only expected (and is required) if **O_CREAT** is one of the flags specified in '*mode*', and specifies the access permissions to be given the new file.

IMPORTANT NOTE: GNO's **read()/write()** functions take a **size_t** count, whereas ORCA's only take unsigned count. When recompiling code with the new GNO libraries, make very certain that any programs that use **read()/write()** do a **#include <fcntl.h>**, or it is likely that your programs will crash.

opendir readdir rewinddir closedir

```
#include <dirent.h>
DIR *opendir(char *filename)
struct dirent *readdir(DIR *dirp)
void rewinddir(DIR *dirp)
closedir(DIR *dirp)
(PPOSIX 1)
```

This family of functions provides a machine-independent way to read a list of files (and information about them) from directories.

opendir() opens the directory specified by *filename* and prepares it for the scan operation. **opendir()** returns a pointer to a structure which is used in the other **dirent** calls.

readdir() takes a **DIR *** as argument and returns information about the next file in the directory. The return value is a pointer to a **dirent** structure (described below).

If you wish to scan the directory again without closing and then reopening the directory, use **rewinddir()**. It resets the scan to the beginning of the directory.

When finished with the directory, call **closedir()**.

```
#define MAXNAMLEN 32      /* maximum filename length */

struct dirent             /* data from getdents()/readdir() */
{
    long          d_ino;    /* inode number of entry */
    off_t         d_off;    /* offset of disk directory entry */
    unsigned short d_reclen; /* length of this record */
    char          d_name[MAXNAMLEN]; /* name of file */
    unsigned short d_namlen; /* length of filename */
};
```

dirent is the structure returned by **readdir()** that contains information about the file. **d_ino** is not used on the Apple IIGS because neither ProDOS nor HFS have the concept of an "inode", but to simulate its use a unique **d_ino** value is returned for each **readdir()** call. **d_off** is the offset in the directory of the current file; the first entry is number 1, the second 2, etc. **d_reclen** specifies the length of the entire **dirent**

structure. **d_name** is a short array containing the filename of the current file read from the directory. **d_namlen** is the length of the string in **d_name**.

More specific information can be obtained by passing **d_name** to the **stat()** system call.
See also: **stat(2)**

needsyno

```
int needsyno(void)
```

This function returns 1 if GNO is operating, and 0 if it is not. Use this function to abort programs that use GNO-specific features, or to allow them to enable non-GNO environment dependent code.

parsearg

```
~GNO_PARSEARG subroutine (4:commandline,4:argptr)
~GNO_PARSEARG(char *commandline, char **argptr)
```

This function will take the command-line passed to a utility and parse it into an **argv,argc** structure like those used in C programs. This was written NOT as a replacement for a C parser, but for use by assembly language programmers writing shell commands.

commandline is the raw command line string as passed by the shell in the X & Y registers. *argptr* is a pointer to an **argv[]**-style array. **parsearg** returns the number of arguments found in the accumulator.

This function ASSUMES that the ByteWorks Memory Manager has been started up and is usable.

This function is based on actual GNO/ME shell (**gsh**) parsing code.

pcreate pbind pgetport psend preceive pdelete preset pgetcount

```
#include <sys/ports.h>
int pcreate(int count)
int pbind(int portid, char *name)
int pgetport(char *name)
int psend(int portid, long int msg)
long preceive(int portid)
int pdelete(int portid, int (*dispose)())
int preset(int portid, int (*dispose)())
int pgetcount(int portid)
```

The Ports IPC mechanism is a very flexible, powerful and efficient method of interprocess communication. A port is a queue that can contain a number of 32-bit values. The size of the port (how many messages it can contain) is specified in the **pcreate()** call.

Creation of a port is done with **pcreate()**. You must specify the size of the port in this call, which must be at least 1 (one). The larger the port, the more data it can hold without blocking a process sending data to the port. **pcreate()** returns a port ID value that must be used in subsequent calls to the Port IPC routines.

A name may be associated with a port; this allows totally unrelated processes to access a port without having to communicate the port ID through some other method, and without knowing the process ID of the other. To bind a name to a port, call **pbind()**. The name argument may be any length, but at most 32 characters are significant. If a name has already been bound to the chosen portid, an error is returned. To

get the portid of a port by its name, use the **pgetport()** call. Pass in the name of the port whose port ID you wish to obtain. If no port has that name, an error is returned. Names are only unbound from a port when a port is deleted.

psend() is used to send a 32-bit datum to a port. If the port is full (that is, if there are more unread messages in the port than are specified in the **pcreate()** call) then the sending process blocks until a message is read from the port. Messages are retrieved from a port using the **preceive()** call. **pgetcount()** returns the number of messages in the port that have not been received; this may be used to avoid blocking on a **psend()** call.

If you wish to clear the contents of a port, say to synchronize communication after an error condition, use the **preset()** call. The arguments to this call are the port ID and the address of a 'dispose' function. Each message in the port, before being cleared, is passed to the dispose function so that appropriate clean-up action may be taken on the data. For example, if the messages correspond to the address of memory blocks obtained with **malloc()**, you could pass 'free()' as the dispose function to automatically deallocate that memory. If you don't wish to take any special action on the data being cleared, pass NULL for the dispose argument.

To destroy a port, make the **pdelete()** call. It accepts the same arguments as **preset()** and they operate as described above. The difference between **preset()** and **pdelete()** is that the latter totally destroys a port; it may no longer be used. **preset()** clears a port's data but leaves the port open for more data transmission.

For an example of the use of ports, see the source code to the print spooling utilities (**lpr**, **lpd**, **FilePort**). These are available from Procyon upon request.

regexp

Compile and execute regular-expression programs. Use 'man regexp' for details.

send receive recvtim recvclr

```
#include <gno/gno.h>
int send(int pid, unsigned long msg);
unsigned long receive(void);
unsigned long recvtim(int timeout);
unsigned long recvclr(void);
```

These kernel functions comprise GNO's message-passing IPC system. Messages are unsigned 32-bit data values. A process sends a message to another by using the **send()** call. You must specify the process ID of the recipient and the message to pass. To receive a message, a process makes the **receive()** call. If no message has been sent to the process, the process sleeps until a message arrives. **recvclr()** is used to clear any pending message a process may have waiting. **recvtim()** is similar to **receive()** but takes a timeout argument, specified in 1/10ths of a second. If no message has been received in *timeout*/10 seconds, **recvtim()** fails and returns -1. The message buffer for a process is only one message deep; any attempt to **send()** a message to a process that already has one queued results in an error. For an IPC system with a deeper queue, see the Ports IPC section.

A **receive()** that is interrupted by a signal will abort and return -1, with **errno** set to **EINTR**.

setenv unsetenv

```
#include <unistd.h>
int setenv(const char *name, const char *value, int rewrite)
void unsetenv(const char *name)
```


Set the value of the environmental variable *name* to be *value*. If *rewrite* is set, **setenv** replaces any current value. The variable is considered 'exported', according to the shell convention for variables. No errors are possible, and the only return code is 0.

unsetenv removes the environmental variable specified by *name* from the variable table. The variable is no longer accessible, and any value that was assigned to that variable is deallocated. No errors are possible, and there is no return value.

statfs

```
int statfs(char *path, struct statfs *buf)
```

Returns information on the filesystem that the file *path* resides on. The information is placed in a structure pointed to by the input argument *buf*. Read **statfs(3)** for more information.

strdup

```
#include <string.h>
char *strdup(const char *str)
```

strdup() creates a copy of the NUL-terminated string pointed to by *str*. It allocates a piece of memory exactly large enough to hold the string with the **malloc()** library function. When you no longer need the copy, dispose of it with **free()**.

See also: **strcpy()**, **malloc()**, **free()**

strsep

```
#include <string.h>
char *strsep(char **stringp, const char *delim)
```

Gets a token from string **stringp*, where tokens are nonempty strings separated by characters from *delim*.

strsep writes NULs into **stringp* to end tokens. *delim* need not remain constant from call to call. On return, **stringp* points past the last NUL written (if there might be further tokens), or is NULL (if there are definitely no more tokens). If **stringp* is NULL, **strsep** returns NULL.

termcap

The termcap library accesses the */etc/termcap* database, which is used to provide terminal- independent support for advanced terminal features, such as various text modes, scrolling regions, cursor movement, and more. Use 'man termcap' for more details.

ttyname

```
#include <unistd.h>
char *ttyname(int fd)
```

Returns the filename of the tty referenced by file descriptor *fd*. If *fd* does not refer to a tty file, NULL is returned. Otherwise, a pointer to the filename (NUL-terminated string) is returned.

tty filenames are in the format ".ttyXX", where XX is a device designator. When porting existing BSD code, take care to watch for code that depends on the existence of a '/' character in the string, as UNIX tty files are in the form "/dev/ttyXX".

The string pointer returned points to a static buffer, and will be overwritten on any further calls to **ttyname**. Copy the string if you wish to preserve it.

unlink

```
int unlink(char *fname)
```

Causes the link file specified by *fname* to be removed. Since GNOME does not yet support symbolic or hard file links, this function operates the same as the **remove()** (or DestroyGS) routine.

NAME

tty - general terminal interface

SYNOPSIS

```
#include <sgtty.h>
```

DESCRIPTION

This file documents the special file `.tty` and the terminal drivers used for user-oriented I/O.

The Controlling Terminal

Every process has associated with it a controlling terminal, which is the terminal the process was invoked from. In some versions of Unix, the controlling terminal association is responsible for job control; this is not so under GNO. A process' controlling terminal is inherited from its parent. By opening the special file `.tty`, a process can access its controlling terminal. This is useful where the input and output of a process was redirected and the process wants to be sure of outputting or getting input from the user at the terminal.

A process can remove the association it has with its controlling terminal by opening the file `.tty` and issuing an

```
ioctl(f, TIOCNOTTY, 0);
```

This is often desirable in server processes.

Process Groups

Every terminal has an associated process group. Any time a signal-generating special character is typed at the terminal, the terminal's process group is sent that signal. Unix systems set process groups using `ioctl()` calls, but under GNO a new interface method is used; process group assignments are controlled with the `JOB CONTROL(2)` routines.

Modes

There are four modes in which terminal drivers operate. These modes control how the driver deals with I/O.

cooked This is the default mode of the terminal driver. If an incoming character is one of the special characters defined in `sgttyb`, `tchars`, or `ltchars`, the appropriate action is performed (see below). This mode also allows for input editing, as input is internally buffered line by line, and data is returned to a reading process only when CR is entered.

cbreak Input is returned on a per-character basis, instead of line by line as in cooked mode. If no data is available, a read will block the calling process. If data is available, a number of characters up to but not exceeding the requested number will be returned. Special characters such as `t_intrc` are not handled, but are passed on to the caller as data.

raw Like *cbreak* mode, except that no input or output processing whatsoever is performed.

Summary of terminal control modes

Due to the colorful history of Unix systems, the data structures used to manipulate terminal modes and settings are separated into four groups. Future revisions of GNO will implement the POSIX termio interface, which consolidates these structures into one place.

sgtty

The basic ioctls use the structure defined in <sgtty.h>:

```
struct sgttyb {
    char  sg_ispeed;
    char  sg_ospeed;
    char  sg_erase;
    char  sg_kill;
    short sg_flags;
};
```

sg_ispeed and sg_ospeed indicate the baud rates for input and output according to the following table. Speed changes that do not apply to a particular piece of hardware are ignored (for instance, the console driver does not access a serial port so all baud rate settings are, in effect, impossible). Also, not all the baud rates supported by a particular device are allowed to be set from this interface.

These symbolic names for the baud rate settings are defined in <sgtty.h>.

B0	0	(hang up dataphone)
B50	1	50 baud
B75	2	75 baud
B110	3	110 baud
B134	4	134.5 baud
B150	5	150 baud
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
B19200		and
EXTA	14	19200 baud
B38400		and
EXTB	15	38400 baud
B57600	6	57600 baud

The sg_erase and sg_kill fields specify the line-editing erase and kill characters. sg_erase is 0x7F (delete) by default, and sg_kill is not currently used.

sg_flags is a bitmapped value that indicates various state settings for the terminal driver (values are in hex).

EVENP	0x80	Use Even parity (serial devices only)
ODDP	0x40	Use Odd parity (serial devices only)
RAW	0x20	Raw mode: wake up on all characters, 8-bit interface
CRMOD	0x10	Map CR into LF; output LF as CR-LF
ECHO	0x08	Echo (full duplex)
CBREAK	0x02	Return each character as soon as typed
TANDEM	0x01	Automatic flow control

RAW and **CBREAK** modes were described above, in Modes.

If the **CRMOD** bit is set, a line feed character is appended to any echoed or ouputted carriage return.

The **ECHO** bit controls input echoing; if enabled, any characters read from the terminal are echoed. This behavior differs slightly from Unix, where input characters are echoed as soon as typed.

TANDEM mode enables automatic software flow control utilizing the special characters `t_startc` and `t_stopc` in `tchars` (below). Whenever the input queue is in danger of overflowing, the system sends `t_stopc`; when the queue has drained sufficiently, `t_startc` is sent. This mode has no effect on the console driver.

Note: `t_startc` and `t_stopc` are used for both directions of flow control; when `t_stopc` is received from a remote system (or user), the terminal stops output, and when `t_startc` is received output resumes. Certain drivers may also require `t_stopc` and `t_startc` to be the same character, in which case one or the other setting will be ignored. See the driver's documentation for details.

Basic Ioctls

Most `ioctl()` calls apply to terminals. They have the form
`#include <sgtty.h>`

`ioctl(int fildes, unsigned long code, void *arg)`

`arg` is usually a pointer to a structure or `int`. The `ioctl` codes that apply to `sgtty` are:

- TIOCGETP** Fetch the basic parameters associated with the terminal, and store in the `sgttyb` structure pointed to by `arg`.
- TIOCSETP** Set the terminal's basic parameters according to the `sgttyb` structure pointed to by `arg`. The input queue is flushed, and the call waits for the output queue to drain before the parameters are changed.
- TIOCSETN** This is like **TIOCSETP**, except there is no delay and the input queue is not flushed.

With the following codes `arg` is ignored.

- TIOCEXCL** Set "exclusive-use" mode. The terminal may not be opened again by any process until all existing references are closed.
- TIOCNXCL** Turns off "exclusive-use" mode.
- TIOCHPCL** When the last reference to the terminal is closed, the terminal line is forced to hang up. This applies only to modem drivers.

With the following codes, `arg` is a pointer to an `int`.

- TIOCGETD** The current line discipline number is stored in the `int` pointed to by `arg`. This value is currently ignored.
- TIOCSETD** The line discipline is set according to the `int` pointed to by `arg`.

TIOCFLUSH The specified queue is flushed. If the value pointed to by `arg` is zero, both the input and output queues are flushed. If the value is `FREAD` (defined in `<sys/file.h>`), the input queue is flushed. If the value is `FWRITE`, the output queue is flushed.

The last few calls permit detailed control of the driver. In cases where an argument is required, it is described. Otherwise, `arg` should be a `NULL` pointer.

TIOCSTI The character pointed to by the argument is placed in the input queue as if it had been typed on the terminal.

TIOCSEBRK Begins a break sequence on the terminal.

TIOCSEBRK Ends a break sequence.

TIOCSDTR The DTR line is turned on

TIOCSDTR The DTR line is turned off

TIOCSTOP Output is stopped as if `t_stopc` had been typed on the terminal.

TIOCSTART If output is stopped, it is resumed as if `t_startc` had been typed on the terminal.

TIOCOUTQ The number of characters in the output queue is returned in the `int` pointed to by `arg`.

FIONREAD The number of characters immediately available for input from the terminal is returned in the `int` pointed to by `arg`. This is the preferred method of non-blocking I/O (checking for the presence of characters without waiting for them).

Tchars

The second structure associated with a terminal defines special characters. The structure is defined in `<sys/ioctl.h>` which is automatically included by `<sgtty.h>`.

```
struct tchars {
    char  t_intrc;      /* interrupt */
    char  t_quitc;      /* quit */
    char  t_startc;     /* start output */
    char  t_stopc;      /* stop output */
    char  t_eofc;       /* end-of-file */
    char  t_brkc;       /* input delimiter (like nl) */
};
```

The default values for these characters are `^C`, `^\\`, `^Q`, `^S`, `^D` and `-1` respectively. A value of `-1` for any of the characters means that the effect of that character is ignored. The stop and start characters may be the same to produce a 'toggle' effect. It is not recommended to set any of the other characters to the same values; the order in which the special characters are checked is not defined, and the results you get may not be what was expected.

The `ioctl` calls that apply to `tchars` are:

TIOCGETC Returns the special characters settings in the `tchars` structure pointed to by `arg`.

TIOCSETC The special characters are set according to the given structure.

Local mode

The third structure in the terminal interface is a local mode word. The various bitfields in this word are as follows (values are in hex):

LCRTBS	0x0001	Backspace on erase rather than echoing erase
LPRTERA	0x0002	Printing terminal erase mode
LCRTERA	0x0004	Erase character echoes as backspace-space-backspace
LTILDE	0x0008	Convert ~ to ` on output (for Hazeltine terminals)
LMDMBUF	0x0010	Stop/start output when carrier drops
LLITOUT	0x0020	Suppress output translations
LTOSTOP	0x0040	Send SIGTTOU for background output (not implemented)
LFLUSHO	0x0080	Output is being flushed
LNOHANG	0x0100	Don't send hangup when carrier drops
LPASSOUT	0x0200	Cooked mode with 8-bit output
LCRTKIL	0x0400	BS-space-BS erase entire line on line kill
LPASS8	0x0800	Pass all 8 bits through on input, in any mode
LCTLECH	0x1000	Echo input control chars as ^?
LPENDIN	0x2000	Retype pending input at next read or input character
LDECCTQ	0x4000	Only ^Q restarts output after ^S
LNOFLSH	0x8000	Inhibit flushing of pending I/O when intr char is typed

The ioctl's used to access the local mode follow. *arg* in all cases is a pointer to an int.

TIOCLBIS The bits of the local mode word specified by *l* bits in the argument are set; this operation is a bit-wise OR.

TIOCLBIC The bits of the local mode word specified by *l* bits in the argument are cleared; this operation ANDs the local mode with the bitwise negation of the argument.

TIOCLSET Sets the local mode word to the value of the argument.

TIOCLGET Returns the local mode word in the int pointed to by *arg*.

Local Special Characters

The fourth terminal structure is another set of special characters. The structure is named *lchars* and is again defined in *<ioctl.h>*.

```
struct lchars {
    char t_suspc;      /* stop process signal */
    char t_dsuspc;     /* delayed stop process signal */
    char t_rprntc;     /* reprint line */
    char t_flushc;     /* flush output (toggles) */
    char t_werasc;     /* word erase */
    char t_lnextc;     /* literal next character */
};
```

Defaults for these characters are ^Z, ^Y, ^R, ^O, ^W, and ^V. As with *tchars*, a value of -1 disables the effect of that character. Only *t_suspc* is currently implemented for the console driver.

The applicable ioctl functions are:

TIOCSLTC sets the local characters according to the `ltchars` structure pointed to by `arg`.

TIOCGLTC retrieves the local characters, storing them in the argument.

Window/terminal sizes

Provision is made for storage of the current window or terminal size along with the other terminal information. This info is recorded in a `winsize` structure, and is defined in `<ioclt.h>`:

```
struct winsize {
    unsigned short  ws_row;      /* rows, in characters */
    unsigned short  ws_col;      /* columns, in characters */
    unsigned short  ws_xpixel;   /* horizontal size, pixels */
    unsigned short  ws_ypixel;   /* vertical size, pixels */
};
```

A '0' in a field indicates that the field value is undefined. '0' is the default when a terminal is first opened. These values are not used by the terminal driver itself; rather, they are for the benefit of applications. The `ioctl` calls for `winsize` are:

TIOCGWINSZ Returns the window size parameters in the provided `winsize` structure.

TIOCSWINSZ Sets the window size parameters. If any of the values differ from the old ones, a `SIGWINCH` signal is sent to the terminal's process group.

FILES

- `.tty`
- `.ttyco` (console driver)
- `.tty*` (user-installed drivers)

SEE ALSO

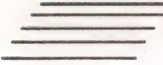
GNO Shell Reference Manual, **stty**(1), **ioctl**(2), **signal**(2)

GNO Shell User's Manual

Version 1.0 Beta Release 2

August 3, 1991

Written by Tim Meekins

 **PROCYON**, INC.

The GNO system and this manual are copyright 1991 by Tim Meekins and Procyon, Inc. The Orca/C run-time libraries are Copyright 1987-1989 by Byte Works, Inc. and used with permission.

This manual was written using AppleWorks GS from Claris Corporation.

Apple IIGS, APW, AppleWorks and GS/OS are registered trademarks of Apple Computer, Inc.

UNIX is a trademark of AT&T Bell Laboratories.

Orca/M and Orca/C are registered trademarks of the Byte Works, Inc.

Important Notice - This is a fully copyrighted work and as such is protected under copyright laws of the United States of America. According to these laws, consumers of copywritten material may make copies for their personal use only. Duplication for any purpose whatsoever would constitute infringement of copyright laws and the offender would be liable to civil damages of up to \$50,000 in addition to actual damages, plus criminal penalties of up to one year imprisonment and/or a \$10,000 fine.

Apple Computer, Inc. MAKES NO WARRANTIES, EITHER EXPRESSED OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED IN SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

This product is sold for use on a *single computer* at a *single location*. For information on obtaining a site license for using multiple copies, contact the publisher.

Procyon, Inc.
1005 N. Kingshighway, Suite 309
Cape Girardeau, MO. 63701
(314) 334-7078

Table of Contents

1. Introduction and Installation
2. Command-Line Editor
3. Command Execution
4. Shell Variables
5. Built-in Commands

Chapter 1

Introduction and Installation

INTRODUCTION

So, you've just purchased the GNO multitasking shell and kernel. Since you have already spent your hard earned cash on this program, you most likely already know a little bit about what the GNO system is all about. Overall, the GNO system consists of two programs in one: the *kernel* and the *shell*. The kernel is the heart of the GNO system, it provides an interface between an *application* and the *operating system*. The kernel's main job is to keep track of processes and perform the actual multitasking. You can think of the kernel as an additional layer placed on top of GS/OS. The only way to communicate with the kernel is through supplied system calls, much like the way applications communicate with GS/OS.

The GNO shell, on the other hand, provides an *interactive* interface to the kernel. The kernel contains a command-line editor in which you, the user, may directly interact with the kernel. The rest of this manual will present the overall scope and usage of the GNO shell. The *GNO Shell Reference Manual* should be used when looking for specific details not easily found in this manual.

STARTING UP THE SHELL

To start the shell, you must first launch the kernel. As the kernel is launched, *it will automatically run the shell for you*. You only need to launch the kernel to start the GNO system. In fact, it is very possible to write your own shell and replace the GNO shell with your own, and still have the power of the GNO kernel available. If you try to run the shell without starting the kernel, you will be presented with an appropriate error message. In fact, all utilities which require GNO specific functions (functions not provided in other shells) will present error messages requiring GNO. The *GNO Kernel Reference Manual* provides further information on determining if the kernel is active. The GNO shell *does* support all utilities written for other shells.

When the shell is started up, it performs a variety of tasks; many of which won't be discussed. The shell does do two things that you should be aware of. First, it will display two lines of "="s. Between these lines, the shell will display the entire contents of the file called *message*, which is in the same directory as the shell. If it does not exist, nothing will be displayed. This file contains pure ASCII text and provides a way for you to post messages or reminders you wish to see each time you time launch the shell. Secondly, the shell will attempt to read a file called *gshrc* from the same directory. This file also contains text, but it is interpreted differently. Instead of displaying the contents of the file, the shell will try to interpret each line of the file as a command. These commands are the same commands you'll be running from the shell, but this special file lets you enter a series of commands for customizing the system. For example, you can set how your prompt will be displayed each time you launch the shell. The following section will discuss using this file to customize your environment. Here is a simple *gshrc* file to give an idea of what one looks like:

```
#
# Sample GNO Shell login file
# last update: 7/9/91
#
#
# set up prefixes
#
prefix 2 /hard1/libraries
prefix 5 /hard1/languages
prefix 6 /hard1/utilities
```



```

#
# Initialize our environment
#
set path="/hard1/utilities"
set prompt="[%h] %S%t%s %C> "
set history=15 savehist=15
set home="/hard1/gno"
set user=appleiiigs
export path prompt history savehist home user
#
# display welcome and time
#
echo " "
echo "Welcome to GNO, special Kansasfest 1991 Edition"
echo " "
echo -n "Today's time:" ; /hard1/utilities/qtime
#
# Move to the home directory
#
cd

```

NOTE: When the shell wants to run a utility command, it searches each directory specified in the path variable. If the utility cannot be found in any directory specified in the path, then it will search the current working directory. Since path is not set until the gshrc file has been read, full pathnames will need to be specified in the gshrc file to run executables. However, *built-in* commands, such as echo, set, and export, may be used. This will be discussed more in the next section.

Once the gshrc file has been read, the shell will display the special shell prompt and await your command. The default prompt is "%", but can be changed to anything else. You are now ready to begin giving commands to the shell to be executed.

CUSTOMIZING THE GNO ENVIRONMENT

As you have most likely noted from the previous section, the gshrc file is a very integral part of the shell. This file allows more than simply customizing the shell; it also provides important information that the shell could not have determined otherwise. The gshrc is simply a shell script containing standard shell commands. The sample shell script in the previous section covers all of the basic settings which you will ever need to be concerned about. The following paragraphs will describe each section of the gshrc file.

The first thing done in the script is to set up several GS/OS prefix numbers which have been reserved for APW/Orca shell environments. These prefixes are not important to the GNO shell, but it is usually a good idea to have them set up properly as there may be existing utilities that assume these prefixes are correct. Prefix 2 contains linkable libraries and macros, prefix 5 contains your language compilers, and prefix 6 contain various utility programs.

Next, the environmental variables must be set up. The GNO shell contains numerous reserved variable names, all of which will not be discussed here. Since all variables have scope, they are only "seen" by the program that sets them. To allow other utilities to access these variables they must be exported using the export command shown in the script.

The most important variable which needs to be set is the `$path` variable. This variable contains a list of pathnames separated by spaces or colons (":"). Because the colons are used as separators, use only '/' in the pathnames, not the optional colon (:) supported by GS/OS. Once the script is finished being read, the shell will search each of these paths for executable files (EXE), system files (S16), and special shell scripts. These filenames will be placed into high-speed look-up tables. When you try to run a program, the shell will quickly search these tables looking for the utility to run. If it finds it, it immediately goes to the proper directory specified in the path and runs the program. If you place new utilities into one of these directories after the tables are built, GNO will not be able to find the utility unless you force the shell to rebuild the tables.

The second variable set is the `$prompt` variable. This allows customization of the command-line prompt. If `$prompt` is not set the shell will default to a '%' prompt. The prompt variable allows you to add things such as current directory name, today's date and time, the current history number, and numerous other options. See the section on variables for the numerous prompt options.

The `history` variable sets how many command-lines are saved in the history buffer and `savehist` determines how many are saved when terminating a session. The `home` variable sets your home or base directory. Type `cd` (change directory) without a directory and it will change the directory to `home`. The `user` variable is used simply in displaying the user name in prompts. When GNO eventually supports multiple users, this variable will be automatically set when "logging in."

This section has now covered those settings which are crucial to the proper operation of the GNO shell. You may set any other variables you need for personal use. You may also use certain utilities which require certain variable settings. If you use several different directories a lot, set variables for them such as `wp` for word-processing and `gr` for graphics programs. Typing `$wp` on the command-line by itself will change the current directory to your word-processing directory without typing in long pathnames. The login script also lets you run any other commands you'd like to run. In the example script, the utility `qtime` was run to display the current time.

Chapter 2

Command-Line Editor

The primary means of communicating with the kernel is using the shell's command-line editor. The command-line editor is very similar to a text editor, except that the domain being worked on is a single line of text instead of a full body of text. The command-line editor only lets you to move forward and backwards through a file, unlike a text editor which provides movement in any direction. Although limited to a line of text, the line can contain 1024 characters, one entire kilobyte of storage. The command-line editor contains numerous editing commands typically only found in text editors.

The command-line editor also supports history buffering. Every command you enter is saved within a buffer (storage area) whose size can be adjusted. When repeatedly entering the same command or a similar command, a single keystroke will display the previous command, additional keystrokes display further commands from the buffer, eventually looping past the oldest command and returning to the most recent history buffer. Any one of these buffers can be edited and executed. History buffers can also be used from one session to another, *ie.*, when quitting from the shell, the current history buffer *can* be saved to the file `*/history`. This is customizable by the user. For more information on customizing the history mechanism, refer to the section on *shell variables*.

Typing normal characters from the keyboard will result in having them displayed on the command-line, one after another. Special editing keys have been provided and are accessed by using special modifier keys. CTRL-char means press and hold the *control* key while pressing another key, "char." OA-char means press and hold the key with the Apple™ logo while pressing another key, "char." Other specially marked keys are used and will be explained.

GNO Shell Command-Line Editor Commands

RETURN	- Typing RETURN will place the command-line in the history buffer then execute it.
LEFT-ARROW	- Moves the cursor to the left one character. Cannot go past beginning of line.
RIGHT-ARROW	- Moves the cursor to the right one character. Won't go past rightmost character.
UP-ARROW	- Replaces the command-line with the previous command
DOWN-ARROW	- Replaces the command-line with the next command
DELETE	- Deletes the character to the left of the cursor.
TAB	- Command/File completion
CTRL-D	- <u>Exits the shell if the command-line is empty.</u>
CTRL-D	- Displays file names matching the current partial name.
CTRL-X	- Clears the entire command-line.
CTRL-Y	- Clears command-line from cursor to end of command-line.
OA-D	- Delete character under the cursor.
OA-E	- Toggles between overwrite and insert mode.
OA->	- Jump to end of command-line
OA-<	- Jump to beginning of command-line
OA-RIGHT	- Jump to end of next word
OA-LEFT	- Jump to beginning of previous word

Chapter 3

Command Execution

After entering a command, pressing the return key will execute that command. By *execute*, we mean that the shell will convert your command into a form recognizable by the GNO kernel providing that the command is entered in such a way that the shell can interpret the meaning. The shell recognizes two types of commands: *built-in commands* which are provided inside the shell, and *executable files* which are programs you have written or were provided on the GNO distribution disks. These commands are loaded from the disk then executed. These commands are either in the current working directory or in special directories which you can set in the `path` variable. The commands stored in files are typically called *utilities* and come in several forms: executables (EXE), applications (S16), and shell scripts (SRC/EXEC). The GNO shell has the additional feature that if you type a directory name instead of a command then the current working directory will be set to the specified path. For example, typing `..` at the command-line will back you up one directory level, thus making the `cd` command obsolete.

The base command consists of two parts: the command name and a list of possibly optional arguments. The command name describes what the shell executes and the arguments are special messages that the shell passes along to the command. These arguments control how the command behaves.

example: `echo II Infinitum`. The first word, `echo`, is the command, which will print the arguments `II` and `Infinitum` to the screen.

In the previous examples, the commands were terminated with a carriage return or an end-of-line designator. These characters are called command terminators in the GNO shell. The shell also contains additional characters which may also be used to terminate a command, but they modify how the command is executed. Since command terminators designate the end of a command, multiple commands may be placed on the command-line provided that they are separated by command terminators. Note, Arnold Schwarzenegger is not command terminator, he is a commando *and* a terminator. This is not the same thing.

To place more than one command on a command-line without changing its behaviour, the `;` (semicolon) terminator is used.

example: `echo -n Today is ; date` - This will print "Today is" on the screen by executing the `echo` command. Once `echo` is completed, the command `date` is then executed which will print the date on the screen. Note, `date` is not supplied with the GNO system.

Notice that the semicolon terminator executes the commands one at a time (synchronously). Since the GNO kernel supports multitasking, it is conceivable (and highly likely!) that you'll want to have several commands executing at the same without waiting for the previous command to finish (asynchronous). This is accomplished by using the `&` (ampersand) command terminator. When a command is terminated with an ampersand, it is placed into the "background." When a process (command) is in the background you may lose some interaction with the process, thus stopping the program may be impossible. When the shell places a process into the background, it will display the special process id of that command on the display surrounded in brackets (`[n]`). If you know the process id, the `kill` command will stop the process for you. You may also use the `ps` command to list all of the current processes that are running in the background (the foreground process, typically `'ps'` itself, will also be displayed.)

example. `grep START *.asm & grep procedure *.pas` - This will search all files ending

with .asm looking for the word START and at the same time it will search for all occurrences of procedure in all files ending with .pas. Since the second terminator does not end with an ampersand, it will be run in the foreground while the first command is running in the background. If an ampersand is placed at the end of the second command, then both commands will be placed in the background and the shell itself will be the foreground process. That means the prompt will again be displayed and you may type commands using the command-line editor while both commands are still running. The output of the search will still be displayed on the screen, and very possible in the same place that you are editing text, so beware. As you've probably noted already, certain characters are reserved as special characters for the command interpreter. To use these characters without the interpreter "grabbing" them, you must quote the characters. The first method is to quote a single character using a backslash ('\') preceding the character to quote. For example, echo \& is a quoted character will display "& is a quoted character" instead of interpreting the & as a command terminator. Single quotes (') and double quotes (") may also be used to quote an entire sequence of characters. The single quote will quote *everything*, and the double quote will quote everything *except* variables. Quotes are also used added spaces which may get "eaten" by the interpreter.

example echo hello world displays hello world, whereas echo "hello world" displays hello world.

example set abc def foo bar sets variable abc to def and foo to bar, wherease set abc "def foo bar" sets the variable abc to the value def foo bar.

WILDCARDS, REGULAR EXPRESSIONS, AND GLOBBING

Regular expression expansion comes in two forms: wildcards, in which you pass a regular expression to a function which returns all matching filenames, and globbing, which replaces globbed filenames in the command-line with a list of all matching filenames.

In simplest terms, you supply the shell with a regular expression and the shell will find all of the files on the disk which match the regular expression.

Regular expressions consist of a sequence of characters which are interpreted in the following ways:

- * - matches 0 or more characters of any type
- + - matches 1 or more characters of any type
- ? - matches exactly 0 or 1 characters
- [abc] - match any one of the characters contained in the brackets.
- [^abc] match in character *not* in the brackets.
- [a-c] match any characters within the range *a-b*

all other characters must be exact matches.

examples:

- hello - exactly matches hello
- * - matches anything
- + - matches anything with at least one character
- *.asm - matches all files ending with .asm
- kern.*.asm - matches anything starting with kern. and ending with .asm
- part[0-9a-f] - matches anything ending with a single digit or the characters a through f.

Chapter 4

Shell Variables

The GNO shell also provides special variables. These can be used to set up shell preferences and may be used in special shell languages, which are forthcoming. Variables are accessed in the command-line by preceding the variable name with a dollar sign (\$). The shell will replace the variable name with the value of the variable before the command is actually executed.

example echo '\$path is set to' \$path The first \$path is not expanded because it is enclosed within single quotes.

Variable values are set using the `set` command. See the built in command references later in this manual for information on using this command. Typing `set` without any arguments will display all of the arguments. `unset` will delete a variable.

Variables also have scope, *ie.* variables created in one program cannot be seen another. In fact, two processes may set variables with the same names and different values and know what their values are. The `export` command will set the export flag on a variable, so when one a process executes another, the variables marked exportable are copied into the child processes' variable space. If this variable is modified, the original variable is not modified unless the child again sets the variable to be exportable. For example, the shell has its variables associated with it, many of which are exported. When a command is executed from the shell, these exported variables are given to the process. If the process modifies these values, the shell's variables will not have these new values unless the process also exports the variables. Note, only shell executable files (text files with command-lines) will get their own variable space. Regular executable commands must specifically request its own variable space using the shell call `PushVariables`, otherwise the command will use its parent's variables, where there parent is typically the shell itself.

The following variables are reserved for special uses in GNO. Many of them describe how the shell will perform certain tasks and they should be set in the `gshrc` file.

HISTORY - Sets the number of history commands remembered. If 0 or not defined then ALL histories are remembered.

SAVEHIST - Sets the number of history commands to save to disk when exiting the shell. If set to 0 or not defined then no histories are saved. When the shell is next run, the histories are read in and the history file killed.

PROMPT - This is a string to print as the prompt. If one is not defined then the default prompt will be "%". Placing the special sequence "%<char>" will also print other useful information. Available sequences include:

%h, %!, or !	Current history number
%t or %@	Current time of day in 12 hour am/pm format
%d or %/	Current working directory
%c, %C, or %.	Trailing component of current working dir.
%S (%s)	Standout/iverse mode On (Off).
%%	Displays a single '%'

HOME - This is the users home directory. In Unix it is shorthanded using a tilde (~). Typing 'cd' w/o a path will set the prefix to HOME.

PATH - PATH contains a list of paths separated by spaces or ':'. Each of these paths will be searched for executables at startup. These will then be installed as utilities which can be run from any directory.

ECHO - If set to a non-null value then commands in a shell executable will be echoed to the standard output device.

0,1,2,.. - Variables numbered 0 through argc correspond to each parameter passed to a shell executable. The first parameter, 0, is the command name itself.

Chapter 5

Built-in Commands

This chapter is meant to briefly introduce the built-in commands supplied with the GNO shell. For more information refer to the man pages for each command which are supplied in the *GNO Shell Reference Manual*.

`cd [pathname]`
changes the current working directory to pathname. If no pathname given then changes to \$home.

`clear`
clears the screen

`echo [-n] [args ...]`
displays the args to the screen. If -n specified then a newline sequence is not sent when finished.

`export [variables ...]`
sets each of the variables as exportable

`history`
displays a history of the previous commands. Use \$history and \$savehist to customize the way that histories are handled.

`kill [-<signal> ] [pid]`
this will kill the process identified by pid. You can alternately send a specific signal to the process.

`prefix prefixnum prefixname`
sets prefix number prefixnum to the path prefixname

`ps`
display all of the current processes in memory. Also displays their status and how long they have been running.

`pwd`
prints the current working directory pathname

`rehash`
Clears the command hash table and re-builds it based on the prompt variable.

`set`
`set [var]`
`set [var=value] ...`
`set [var] [value]`
set, by itself, displays all variable settings. Set with one variable displays the value of the variable. Set with a variable followed by value sets that variable to the value, alternately, an argument of var=value may also be used.

`unset [var...]`
deletes each of the specified variables

which [command]

displays which command is executed. For example, what if you created a file in the current directory called 'echo', a file called 'echo' in the path variable, plus the built-in command 'echo' already exists. Which 'echo' will get executed? Running "which echo" will tell you which 'echo' will be executed.

NAME

intro - introduction to commands and application programs

DESCRIPTION

This section describes, in alphabetical order, commands available for GNO. Certain distinctions of purpose are made in the headings. For example, BUILT-IN UTILITIES are those commands which are part of the GNO shell and not contained in stand-alone files.

Manual Page Command Syntax

Unless otherwise noted, commands described in the **SYNOPSIS** section of a manual page accept options and other arguments according to the following syntax and should be interpreted as explained below:

name [-*option*...] [*cmdarg*...]

where:

[]	Surround an <i>option</i> or <i>cmdarg</i> that is not required.
...	Indicates multiple occurrences of the <i>option</i> or <i>cmdarg</i> .
<i>name</i>	The name of an executable file.
<i>option</i>	(Always preceded by a "-".) <i>noargletter</i> ... or, <i>argletter</i> <i>optarg</i> [,...]
<i>noargletter</i>	A single letter representing an option without an option-argument. Note that more than one <i>noargletter</i> option can be grouped after one "-".
<i>argletter</i>	A single letter representing an option requiring an option-argument.
<i>optarg</i>	An option-argument (character string) satisfying a preceding <i>argletter</i> . Note that groups of <i>optargs</i> following an <i>argletter</i> must be separated by white space and quoted.
<i>cmdarg</i>	Path name (or other command argument) <i>not</i> beginnning with a "-", or "-" by itself indicating the standard input.

SEE ALSO

GNO Shell User's Manual

NAME

`cd` - change working directory

SYNOPSIS

`cd [ directory ]`

DESCRIPTION

If *directory* is not specified, the value of the shell variable `$home` is used as the new working directory. If *directory* specifies a complete path starting with `/`, `..`, `...`, `~`, *directory* becomes the new working directory. If neither case applies, *cd* tries to find the designated directory in the current working directory.

SEE ALSO

`pwd(1)`, `gsh(1)`
GNO Shell User's Manual

AUTHOR

cd was written by Tim Meekins for the GNO shell.

NAME

clear - clear terminal and home cursor

SYNOPSIS

clear

DESCRIPTION

clear sends a FormFeed (12) to the standard output which on a terminal screen will clear the screen and home the cursor. On a printer this will form feed the paper.

SEE ALSO

GNO Shell User's Manual

AUTHOR

clear was written by Tim Meekins for the GNO shell.

NAME

echo - echo arguments

SYNOPSIS

echo [**-n**] [arg] ...

DESCRIPTION

The *echo* command writes its arguments separated by blanks and terminated by a new-line on the standard output. It also understands C-like escape conventions:

\b	backspace
\f	form-feed (clear screen)
\n	new-line (generates a carriage return and line-feed)
\r	carriage return
\t	tab
\0n	where <i>n</i> is an 8-bit character whose ASCII code is the 1-, 2-, or 3-digit <i>decimal</i> number representing that character.

options include:

-n Suppresses the newline character which terminates the output.

SEE ALSO

gsh(1)
GNO Shell User's Manual

CAVEATS

The shell will try to expand the "\ " unless the arguments to echo are surrounded by *single* quotes (')

Also, most UNIX systems use octal number representations for the character escaping. The GNO version is using *decimal*.

AUTHOR

echo was written by Tim Meekins for the GNO shell.

NAME

`export` - tag variable as exportable

SYNOPSIS

`export` file ...

DESCRIPTION

`export` marks each file as exportable. When a shell script is run or a new variable stack frame is created, then those variables marked exportable will be copied to the new variable stack frame. Exporting a variable inside of a shell script will mark it so that the variable will remain after the shell script exits.

SEE ALSO

`gsh(1)`, `set(1)`, `unset(1)`
GNO Shell User's Manual

CAVEATS

When a new variable stack frame is created, exported variables are copied into it. These copied variables are *not* marked as exportable. They must again be exported for their values to be passed back up the variable stack frame.

AUTHOR

`export` was written by Tim Meekins for the GNO shell.

NAME

history - command line history

SYNOPSIS

history

DESCRIPTION

history displays the entire history list of command-lines stored in memory. If **\$history** is set to an integer, then only that number of histories will be saved. Similarly, **\$savehist** determines how many command histories are stored to disk when a session is terminated.

SEE ALSO

gsh(1)

GNO Shell User's Manual

AUTHOR

history was written by Tim Meekins for the GNO shell.

NAME

kill - terminate a process

SYNOPSIS

kill [-signo] PID ...

DESCRIPTION

The *kill* command send signal 15 (terminate) to the specified processes. This will normally kill processes that do not catch or ignore the signal. The process number of each asynchronous process started with **&** is reported by the shell (unless more than one process is started in a pipeline, in which case the number of the last process in the pipeline is reported). Process numbers can also be found by using *ps*(1).

If a signal number preceded by - is given as first argument, that signal is sent instead of terminate. In particular "kill -9 ..." is a sure kill.

SEE ALSO

gsh(1), **ps**(1)

kill(2), **signal**(2)

GNO Shell User's Manual

GNO Kernel Reference Manual

AUTHOR

kill was written by Jawaid Bazyar for the GNO shell.

NAME

prefix - sets standard GS/OS prefix numbers

SYNOPSIS

prefix number pathname

DESCRIPTION

The *prefix* command assigns *pathname* to the GS/OS prefix *number*. *Number* must be a legal GS/OS prefix number (which is currently 0-31). *Prefix* will check for the existence of *pathname* before assigning it to *number*.

SEE ALSO

gsh(1), ***cd***(1)
GNO Shell User's Manual

AUTHOR

prefix was written by Tim Meekins for the GNO shell.

NAME

`pwd` - working directory name

SYNOPSIS

`pwd`

DESCRIPTION

The `pwd` command prints the path name of the working (current) directory.

SEE ALSO

`cd(1)`

GNO Shell User's Manual

AUTHOR

`pwd` was written by Tim Meekins for the GNO shell.

GNO Kernel Reference Manual

Version 1.0 alpha

August 10, 1991

Written by Jawaid Bazyar

Copyright 1991 by Procyon, Inc. All Rights Reserved

The **GNO** Multitasking Environment for the Apple IIgs
Copyright 1991, Procyon Software and Tim Meekins

This program contains material from the ORCA/C
Run-Time Libraries, copyright 1987-1989
by Byte Works, Inc. Used with permission.

distributed by and leased to Procyon, Inc.
Please direct all inquiries to:

Procyon, Inc.
1005 N. Kingshighway, Suite 309
Cape Girardeau, MO 63701
(314) 334-7078

For on-line technical assistance, contact:

GEnie : J.BAZYAR
Internet : bazyar@uiuc.edu

written by:

Jawaid Bazyar	Kernel
Tim Meekins	Shell
Derek Taubert	Kernel support
Greg Thompson	Shell support

With many thanks to:

Bill Gulstad, Rob Knauerhase, and everyone in UIUC Apple II R&D and the A2RD Technical Discussion group
Dave Lyons, for tolerating my endless questions since the beginning of time, and for getting me started in IIgs programming.
Mike Westerfield, for your technical assistance and all your work in making decent development tools for the IIgs.

Special thanks to:

Matt Deatherage, for telling me it couldn't be done, so that I could do it.

Table of Contents

Chapter

- 0. Introduction
- 1. Modifications to GS/OS to support the multitasking environment
 - A. Prefixes
 - B. Open Files
 - C. True TTY Support
 - D. ToolBox modifications
- 2. Orca™ Compatible shell calls
- 3. Process Management
 - A. Process IDs
 - B. `fork()` and `exec()`
 - C. `kill()`
 - D. KVM routines (a slight misnomer)
- 4. Interprocess Communication
 - A. Semaphores (screate, swait, etc).
 - B. Software Signals
 - C. Pipes
- 5. GNO Compliance
 - A. Detecting GNO
 - B. Terminal I/O
 - C. Stack usage
 - D. Disk I/O

Appendices

- A. Kernel system call manuals

PREFACE

This is a draft copy of the *GNO Kernel Reference Manual* for the *alpha* version of the GNO kernel and shell. The is to forewarn you that there may exist differences between this manual and the system as it exists in your computer. At this moment in time, the GNO software is very dynamic and is changing daily. This manual may at times mention features which are forthcoming. When possible, these are marked with a *Alpha* release note.

INTRODUCTION

Computers are tools. The flexibility of a tool determines how useful it is. Early computers were much like the one this software was written for: the Apple IIgs. They could only run one program at a time, and their usefulness was limited to what the particular program the user was executing offered. In the late 1960's, a team of researchers at AT&T began developing the Unix* operating system. Unix's design was partially based on the premise that most programs are I/O bound, i.e. most of the time the program executes is spent waiting for user input or other I/O operations. While one program is waiting for I/O, why not allow another program to execute? This is what they did, and the result was one of the most successful computer projects ever.

The Apple IIgs, like the Macintosh it is modelled after, provides very limited multitasking abilities in the form of desk accessories (NDAs). The programs in the NDA menu are available in whatever application you use as long as it follows Apple's guidelines. However, there are many graphics based programs that don't support NDAs, and in addition there is a wealth of software that has been developed for the Byte Works' Orca environment. This environment is mainly text-based, and thus makes access to NDAs impossible. As if that wasn't enough, it's very difficult to write an NDA to allow the application to keep running. So the benefits are lost, and we're back at ground zero.

Enter the GNO system. What was once just dreamed about is now a reality. GNO provides an environment that is almost entirely compatible with software developed for the Orca environment. But GNO provides a wealth of new abilities, lots of new ground for developers and users alike.

Before we begin describing, we'd like to rebuttal those who say such a multitasking system isn't possible on the IIgs. Obviously it is: you hold it in your hands. Some say the IIgs isn't powerful enough to make multitasking useful. We point out that the IIgs is much more powerful than the first computers Unix was designed to run on (they only had 64K of real memory! and were 16 bit machines). Some ask why you'd ever need to run more than one program at once. These are the same people who asked why we'd ever need more than 64K of memory, or more than 140K of storage on disks. (end soapbox)

First off, GNO provides pre-emptive multitasking. Many programs can be executing at the same time- each is called a 'process'. Each process is allowed to run for a short period of time (1/60th of a second on average). When its time runs out, the current process is set aside and another one chosen to run next. This cycle continues until there are no more processes left (i.e. when you exit GNO). Starting up processes to run 'in the background' is a simple matter of adding a few characters to the Orca shell commands you're already familiar with. As mentioned before, most existing utilities are compatible with GNO.

GNO provides a new shell that takes full advantage of the multitasking ability provided. The most important feature of the shell is process control (starting them up, killing them, and suspending them). But the shell also provides power never before seen on the Apple IIgs. The ability to choose files by 'wildcard' has been around for a while, but GNO takes this to a new level with 'regular expressions', a very powerful yet simple programming language. Other benefits of the GNO shell are too numerous to mention. (see the chapter on the Shell for details).

In addition to being compatible with the Orca system, GNO is a very powerful programming environment. Available to the programmer are all the calls needed to control processes, support Inter-Process Communication, and other tools needed in a multitasking environment. (see the Chapter on Kernel calls for details).

GNO also boasts the first completely consistent method for accessing serial and console I/O. The IIgs TextTools have been incredibly enhanced to provide a truly all-encompassing interface for serial, console, and IPC applications. Imagine being able to attach terminals to your GS, and have a useful shell in each one. Multiuser BBSs, remote dial-ups, UUCP or SLIP that doesn't take over your computer- the applications are endless!

GNO works wonderfully with accelerator boards- in fact, software can work *faster* under the GNO environment than normal! Because programs no longer take processor time waiting for an I/O event to occur,

they also aren't constantly accessing the I/O memory- accelerator boards have to slow down and flush their caches when I/O is accessed. As an example, a GS/OS telecommunications program was 10%-15% faster during file transfers.

With all this talk of shell utilities, have desktop users (users of programs like Prizm*) been left behind? Absolutely not. GNO doesn't allow more than one desktop program to run concurrently, but it DOES let you run a desktop program with as many text applications as you like. In other words, no functionality is lost from the IIGs by using GNO.

Finally, the GNO environment comes with a large number of free utilities that bring some of the power of a Unix system to the IIGs. Also, a number of programming libraries are included that make it easy to port programs from Unix or MS-DOS systems to the IIGs. A whole new world of software awaits the IIGs.

GNO by itself does not require any resources above what you normally use in your development environment. However, if you're going to be running several large programs at once (for instance, background communications software or multiple compiles in the background), then you're definitely going to need as much RAM as you can afford. Also, since the IIGS is by no means a speed demon, an accelerator card is recommended.

The next several chapters describe the resources provided by the GNO system in a general manner. They deal mainly with concepts, and exist to provide an overview and a sort of 'textbook' understanding of how the system as a whole works. At the end of the manual you will find a canonical list of all the functions that GNO provides: the system calls. These are presented in a Unix *manpage* format. This means you will also find these call descriptions on the GNO distribution disk. They are provided in this way so that one need not constantly refer to a printed manual. It also provides a method of impeccable indexing, via the *grep* program. The manpages detail exactly what the programmer must do in order to use the system calls, and also exactly what effects the call has.

NOTE: The material in this manual is of a technical nature. It is intended mainly for programmers who wish to write applications which take advantage of GNO's features. For information on using the shell, see the GNO Shell User's Manual.

Chapter 1: Modifications to GS/OS

The GNO system modifies the behavior of a number of GS/OS calls in order to allow many programs to execute concurrently, and to effect new features. The changes are done in such a way that old software can take advantage of these new features without modification. Following is a complete description of all the changes affected. Each section has details in text, followed by a list of the specific GS/OS or ToolBox calls effected.

Mutual Exclusion in GS/OS and ToolBox calls

The Apple IIGS was not designed as a multitasking machine, and GS/OS and the ToolBox reflect this in their design. The most notable problem with making a multitasking environment work on the IIGS is the use of global (common to all processes) information, such as prefixes and direct page space for tool sets which includes information like SANE results, QuickDraw drawing information, etc. In most cases we've corrected these deficiencies by keeping track of such information on a per-process basis, that is, each process has it's own copy of the information and changes to it do not affect any other process' information.

However, there were many other situations where this could not be done. Therefore, task switching is turned off during all GS/OS and ToolBox calls (except for User Tools and the TextTools). This was the best solution for this problem, and doesn't hurt system performance much.

Prefixes

Normally under GS/OS there are 32 prefixes, and these are all under control of the current application. GNO extends this concept to provide each process with it's own copies of certain prefixes (0, 1, and 9). When a process modifies one of these prefixes via the GS/OS SetPrefix call, it modifies only it's own copy of that prefix- the same numbered prefixes of any other processes are not modified.

```
$2004 ChangePath
$200B ClearBackupBit
$2001 Create
$2002 Destroy
$200E ExpandPath
$2006 GetFileInfo
$200A GetPrefix
$2010 Open
$2005 SetFileInfo
$2009 SetPrefix
```

(Note: The corresponding ProDOS-16 interface calls are also supported).

Open File Tracking

Previously, a major problem with the way GS/OS handled open files was this: a Desk Accessory (or a background program of any sort) could open a file and have it closed without it's knowledge by the main application program. This presented all kinds of problems for desk accessory authors. Apple presented a partial solution with System Software 5.0.4, but it wasn't enough for a true multitasking environment. GNO keeps track of exactly which process opened which file. It also discontinues the concept of a global File Level, opting instead for a per-process File Level. To summarize, if process A opens several files, and process B subsequently makes a Close(0) call, only files opened by process B and above process B's current File Level will be closed. Process A's files are unaffected, even if process A had opened the same files.

In addition to this behavior, when a process terminates in any manner all files that it currently has opened will be closed automatically. This prevents problems of the sort where a program under development terminates abnormally, often leaving files open and necessitating a reboot.

The **Flush()** GS/OS call is *not* modified in this manner. The effects of a **Flush(0)** call are basically harmless, but it will decrease system performance if executed often.

```
$2010 Open
$2014 Close
$201B GetLevel
$201A SetLevel
```


Quitting applications

The GS/OS Quit call has been modified to simply terminate the calling process. It no longer supports transferring to other applications or maintenance of a return stack, neither of which are necessary with the functionality of the `exec()` system call (which is how all applications are launched under the GNO system). Note that this will possibly change in the future to support the Finder*.

Chapter 3: Process Management

Before discussing process management using Kernel calls, it would be wise to define just exactly what we refer to when we say *process*. A *process* is generally considered to be a program in execution. "A program is a passive entity, while a process is an active entity." (Operating Systems Concepts p.73, Silberschatz and Peterson, Addison-Wesley, 1989). The concept of process includes the information a processor needs to execute a program (such as the program counter, register values, etc).

In order to execute multiple processes, the operating system (GNO and GS/OS in this case) has to make decisions about which process to run and when. GNO supports what is termed *pre-emptive multitasking*, which means that processes are interrupted after a certain amount of time (their time slice), at which point another process is allowed to run. The changing of machine registers to make the processor execute a different process is called a *context switch*, and the information necessary for this is called its *context*. The kernel maintains a list of all active processes, and assigns time slices to each process according to their order in the list. When the kernel has run through all the processes, it starts again at the beginning of the list. This is called *round-robin scheduling*. Under certain circumstances, a process can actually execute longer than its allotted time slice because of a GS/OS or ToolBox call. In these cases, as soon as the system call is finished the process is interrupted.

Processes can give up the rest of their time slice voluntarily (but not necessarily explicitly) in a number of ways, terminal input being the most common. In this case, the rest of the time slice is allocated to the next process in line (to help smooth out scheduling). A process waiting on some event to happen is termed *blocked*. There are many ways this can happen, and each will be mentioned in its place.

An important item to remember is the *process ID*. This is a number which uniquely identifies a process. The ID is assigned when the process is created, and is made available for reassignment when the process terminates. A great many system calls require process IDs as input. Do not confuse this with a *userID*, which is a system for keeping track of memory allocation by various parts of the system, and is handled (pardon the pun) by the Memory Manager toolset. Finally, do not confuse Memory Manager *userID*'s with Unix *userID*'s- numbers which are assigned to the various human users of a multiuser machine.

There are two methods for creating new processes- the system calls `fork()` and `exec()` (specifics for calling these functions and others is in the appendix "System calls"). *fork* starts up a process which begins execution at an address you specify. *exec* starts up a process by loading an executable file (S16 or EXE). *fork* is used mainly for use inside a specific application, such as running shell built-ins in the background, or setting up independent entities inside a program. Forked processes have some limitations, due to the hardware design of the IIGS. The parent process (the process which called *fork*) must still exist when the children die, either via `kill()` or by simply exiting. This is because the forked children share the same memory space as the parent; the memory the children execute from is tagged with the parent's *userID*. If the parent terminated, the children's code would be deallocated and likely overwritten. A second caveat with *fork* is the difference between its Unix counterpart. Unix *fork* begins executing the child at a point directly after the call to *fork*. This cannot be accomplished on the IIGS because virtual memory is required for such an operation; thus the need to specify a *fork* child as a C function. Note that an appropriately written assembly language program need not necessarily have these restrictions. When a process is forked, the child is given its own direct page and stack space under a newly allocated *userID*, so that when the child terminates this memory is automatically purged.

exec is used when the process you wish to start is a GS/OS load file (file type S16 and EXE). *exec* follows the procedure outlined in the GS/OS Reference Manual for executing a program, and sets up the new program's environment as it expects. After *exec* has loaded the program and set up its environment, the new process is started and *exec* returns immediately.

Both *fork* and *exec* return the process ID of the child. The parent may use this process ID to send *signals* to the child, or simply wait for the child to exit; indeed, this is the most common use. The system call to accomplish this is *wait*. It takes no parameters, and blocks the caller. Whenever a child process terminates or is stopped (see the section on Signal management), *wait* unblocks the caller and returns the process ID of the affected child.

exec is actually implemented as two other system calls- *fork*, and one called *execve*. *execve* loads a program from an executable file, and begins executing it. The current process' memory is deallocated. The shell uses a `fork()/execve()` pair explicitly, so it can set up redirection and handle job control. See the appropriate manpages for details.

Chapter 5: GNO Compliance

For a program to work effectively under the GNO system, certain rules must be followed. Most of these rules boil down to one underlying concept - never directly access features of the machine. Always use GS/OS, the ToolBox, or GNO to accomplish what you need. We have taken great care to provide the sorts of services you might need, such as checking for input without having to wait for it. GNO compliance isn't just a matter of trying to make applications work well under the environment- it ensures that those applications stay compatible, no matter what changes the system goes through. Below are summarized the points you must consider when you're writing a GNO compliant application.

Detecting the GNO Environment

If your applications requires the GNO Kernel to be active (if it makes any kernel calls), you can make sure this is so by making a *kernStatus* call at the beginning of your program. The call will return no error if the kernel is active, or it will return an error code of \$0001 (Tool locator - tool not found), in which case the value returned will be invalid. The call returns a 1 if no error occurs, but the value returned will be indeterminate if the kernel is not active, so you should only check for an error (*toolerror()* or *_toolErr* in C, the value in the A register in assembly).

You can also determine the current version of GNO by making the *kernVersion* call. The format of the version number returned is the same as all the standard tools, and is documented in the Toolbox Reference.

Terminal I/O

The Apple II has always been lacking in standardized methods for reading keyboard input and controlling the text screen. This problem was compounded when Apple stopped supporting the TextTools in favor of the GS/OS Console Driver. The Console Driver has a number of problems that prevent it from being a good solution. There is high overhead involved in using it. It must be accessed like a regular file, which means any I/O on it must filter through several layers before being handled. There is no provision for patching the low-level routines, so it can be used over a modem or in a desktop program. And GS/OS must be called to access it, which means that while a Console Driver access is occurring, no other processes can execute (see Chapter 1).

GNO ignores the console driver and replaces the TextTools with a high-performance, very flexible generic terminal control system. GNO directly supports the console (keyboard and screen), as well as the serial ports, as terminals. In order for a user program to take advantage of these features and to be GNO compliant, you must do terminal I/O only through the TextTools. Calls to the GS/OS console driver will not crash the system, but they will make other processes stop until the call is completed. You must not get input directly from the keyboard latch (memory location \$E0C000), nor may you write directly to the screen memory. The TextTools Terminal I/O system has been designed so you don't have to do either of these things. If you need to wait for keyboard input without stopping your application, you can make the appropriate Control call (see the chapter on Terminal I/O) to put the tools in the proper mode.

(Note: late breaking news has it that Apple has modified the Console Driver to allow patching of the low-level I/O routines. This means that Console Driver support can very likely be added to GNO while still maintaining current functionality. We salute this move on Apple's part- keep up the good work!)

Stack Usage

Stack space is at a premium on the IIs. Process stacks can only be located in Bank 0- a total of 64K. This theoretical limit doesn't apply, however, as GS/OS and other bits of system software reserve a large chunk of this without any way to reclaim it. There is approximately 48K of usable stack space. This space also has to be shared with direct page space for Tools and certain types of device drivers, however. For a program to be GNO compliant, stack usage analysis must be done and acted upon. Use of the stack should be minimized, in order that many processes can coexist peacefully. From experience we've found that 1K usually suffices for well-written C applications, and at a maximum 4K can be allocated.

Assembly language programs are very efficient when it comes to use of the stack. The 1K provided by default to applications is usually more than enough for assembly language programs. However, C programs can use up tremendous amounts of stack space, especially if recursion is employed or string manipulation is done without concern for stack usage. Below are some hints to keep stack usage at a minimum.

- o Avoid use of large local arrays and character strings. Instead, dynamically allocate large structures such as GS/OS strings with *malloc()* or the Memory Manager.

- o Try not to use recursion unless absolutely necessary. All recursive functions can be rewritten using standard loops and creative programming. This is a good general programming rule- your program will run faster.
- o Orca/C generates 8K of stack by default, in case the desktop is started up. Since GNO compliant programs generally will not be desktop-based, make sure you judge how much stack your program will require and use the `#pragma stacksize` directive to limit how much stack space Orca/C tries to allocate for your program.

Disk I/O

Since the Apple IIs doesn't have coprocessors to manage disk access and the serial ports, either of these requires the complete attention of the main 65816 processor. This can wreak havoc in an environment with slow disks or high-speed serial links, as accessing disks usually results in turning off interrupts for the duration of the access. This situation is lessened considerably with a DMA disk controller, such as the Apple High Speed SCSI or CV Technology RAMfast. But this isn't as bad as it sounds; the IBM PC and Macintosh also suffer from this problem, and the solution is robust programming. Make sure your communications protocol can handle errors where expected data doesn't arrive quite on time, or in full. The best solution would be an add-on card with serial ports and an on-board processor to make sure all serial data was received whether or not the main processor was busy (this is a hint to some enterprising hardware hacker, by the way).

Yet another concern for GNO applications is file sharing. GS/OS provides support for file sharing, but it is up to the application author to use it via the `requestAccess` field in the `open()` call. GS/OS only allows file sharing if all current references to it (other instances of the file being opened) are read-only. GNO authors should use read-only access as much as possible. For example, an editor doesn't need write permission when it's initially reading in a file. Note that the `fopen()` call in C supports read-only mode.

INTERMEDIATE ALPHA RELEASE NOTES (August 10, 1991)

Known Bugs

GNO crashes if AppleTalk is turned on

Some programs (most notably editors like emacs and ROSE) lose characters due to conflicts with the GNO keyboard buffering scheme.

There is a random error that occasionally closes files when loading a program. This most often manifests itself when running programs in the gshrc file, but can wreak havoc elsewhere. Try 'ls & ls'.

I've had trouble with an old version of macgen not being able to find files.

Features

GNO now supports restartable programs, if they terminate with a Quit call and set the appropriate flags (the way Apple recommends it be done). Programs that I know support this: emacs. All applicable Shell Utilities will also support this at the next release.

About this special alpha version of GNO:

We have decided to release GNO in this partially-finished state for a couple reasons:

1) We want as much outside user commentary and suggestions as we can get. The feedback we receive on this alpha version will to a large degree determine the course of GNO, regarding functionality and features.

2) We want to make a statement that this project is here, it's real, it's NOW. We want to make this clear to Apple and to other development community members. We want your support, and the best way of achieving that is showing what we have, and asking you where we should go.

Thanks for trying out the GNO system. I believe it can be a tremendous boost to the Apple II community; and it starts with you.

Sincerely,
Jawaid Bazyar

p.s. Remember - if someone comes up to you on the street and offers you some IBM, Just Say
GNO!

NAME

exec - load an executable file and launch as a new process

SYNOPSIS

```
#include <kerntool.h>
```

```
int exec(char *pathname, char *cmdline);
```

DESCRIPTION

exec is an alternative method for launching a new process under GNO. It is a combination of the *fork()* and *execve()* calls, and in fact is written in terms of them. The difference between *exec* and *execve* is that *exec* returns in all cases- it is a simplified interface for those who don't need to affect process information before a new executable begins (such as process group IDs or I/O redirection information).

The algorithm for *exec* (simplified somewhat) is as follows:

```
void exec2() {  
    return execve()  
}  
int exec() {  
    return fork(exec2);  
}
```

RETURN VALUE

exec returns the process ID of the child, or **YSERR** (-1) if an error occurs in the *fork()* call.

BUGS

See **BUGS** in *execve(2)* and *fork(2)*.

If an error occurs in the *execve* phase of this call, it can't be detected by normal means. The *wait* system call must be used to get the return code of the process, which will be **YSERR** (-1) if the *execve* fails.

SEE ALSO

execve(2), *fork(2)*, *wait(2)*

NAME

execve - replace current process with an executable image from a file

SYNOPSIS

```
#include <kerntool.h>
```

```
int execve(char *pathname, char *cmdline);
```

DESCRIPTION

execve is the preferred method for loading program files to be executed under the GNO system. A new userID is allocated for the process, and the GS/OS System Loader is used to bring the executable file specified by *pathname* into memory. *pathname* can be a partial or complete path. The executable loaded replaces the executable associated with the current process.

If the executable file does not contain an OMF Stack Segment (SEGKIND = \$12), a default stack of 1024 bytes is allocated to the process. The direct-page pointer is set to the bottom of the stack memory (for C programs this is irrelevant).

The parameter *cmdline* is the list of arguments to be passed to the new process (a copy is actually passed). C programs parse *cmdline* automatically, and the individual pieces can be accessed through the *argc/argv* arguments to *main()*. *cmdline* can be accessed from assembly language through the X (high-order word of *cmdline*) and Y (low-order word) registers. However, if the executable file is of filetype S16 (\$B3), the *cmdline* argument is ignored and the X&Y registers are set to null (i.e. the command line is only passed to an EXE executable). The 8 characters "BYTEWRKS" are prepended to *cmdline* before being passed to the process (this is the same identifier used by the Orca shell). This *Shell Identifier* distinguishes the GNO and Orca environments from others that don't support the full range of shell calls, and can be accessed from C with the library function *shellid()*. The A register is set to the userID allocated for the process.

GS/OS prefixes 1 and 9 are set to the pathname of the directory containing the executable file; if the length of the path exceeds 64 characters prefix 1 is set to the null prefix (length 0). The following information is inherited by the child: current machine state, controlling TTY, and process group ID.

Signal handling information is set to defaults. Any signals in the parent's queue are not passed to the child, and the child is started with no signals blocked. The child begins with no open files.

RETURN VALUE

A successful *execve* does not return, as the current executable is replaced with the one specified in the call. If for some reason the call fails, *execve* returns **SYSERR** (-1).

BUGS

Orca/C currently ignores any stack space allocated for it by the GS/OS Loader (which *execve* calls) or by default in *execve*. Stack space in Orca/C programs is determined by code in the .root object file, and can be set with the **#pragma stacksize** directive. Read the section on GNO Compliance in the *GNO Kernel Reference Manual* for more information on this topic.

SEE ALSO

exec(2), *fork*(2), *wait*(2)

THE UNIVERSITY OF CHICAGO

DEPARTMENT OF THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

THE HISTORY OF ARTS

NAME

getpid - return process ID of caller

SYNOPSIS

#include <kerntool.h>

int getpid(void);

DESCRIPTION

The process ID is a unique value associated with a process, and is needed for many system calls. Note that GNO pid's cover only a small range of numbers, as opposed to *nix pid's which range from 0-65535. Some programs use *getpid* to seed random number generators. A much better approach on the IIGS is to use the horizontal and vertical positions of the electron gun, which can be obtained by reading the word value at absolute memory location **\$E0C02E**.

RETURN VALUE

The process ID of the caller is returned. No errors are possible.

NAME

kill - send a signal to a process

SYNOPSIS

```
#include <kerntool.h>
```

```
int kill(int pid, int sig)
```

DESCRIPTION

kill is used to send a signal to a process or group of processes. Signals are software interrupts; they act just like hardware interrupts and can also be used for basic IPC (Inter-process communication). The various signals are described in *signal(3)*.

sig can be a signal number, or it can be 0, in which case no signal is sent, but error checking is done (this can be used to verify a process ID). If **sig** has been blocked (*sigblock(2)*), the signal is deferred until it is unblocked, and **kill** returns immediately. Any previously pending signals of the same **sig** are lost (i.e. signals are not stacked).

If **pid** is 0, the signal is sent to all processes with the same process group ID as the caller, except for system processes.

Processes may signal themselves, in which case the signal handler is invoked immediately (if installed).

RETURN VALUE

Normally **kill** returns **SYSOK** (0). The following conditions can cause **kill** to return **SYSERR** (-1):

- pid** does not correspond to an existing process
- sig** specifies an invalid signal number

BUGS

Currently there is no way to distinguish the types of errors that can cause **kill** to fail. There is no obvious way to return such a value from the toolbox routines that are the kernel (and are written in C).

11. 11. 1941

Amelk, 1997; 1998; 1999; 2000; 2001; 2002; 2003; 2004; 2005; 2006; 2007; 2008; 2009; 2010; 2011; 2012; 2013; 2014; 2015; 2016; 2017; 2018; 2019; 2020; 2021; 2022; 2023; 2024; 2025; 2026; 2027; 2028; 2029; 2030; 2031; 2032; 2033; 2034; 2035; 2036; 2037; 2038; 2039; 2040; 2041; 2042; 2043; 2044; 2045; 2046; 2047; 2048; 2049; 2050; 2051; 2052; 2053; 2054; 2055; 2056; 2057; 2058; 2059; 2060; 2061; 2062; 2063; 2064; 2065; 2066; 2067; 2068; 2069; 2070; 2071; 2072; 2073; 2074; 2075; 2076; 2077; 2078; 2079; 2080; 2081; 2082; 2083; 2084; 2085; 2086; 2087; 2088; 2089; 2090; 2091; 2092; 2093; 2094; 2095; 2096; 2097; 2098; 2099; 2100; 2101; 2102; 2103; 2104; 2105; 2106; 2107; 2108; 2109; 2110; 2111; 2112; 2113; 2114; 2115; 2116; 2117; 2118; 2119; 2120; 2121; 2122; 2123; 2124; 2125; 2126; 2127; 2128; 2129; 2130; 2131; 2132; 2133; 2134; 2135; 2136; 2137; 2138; 2139; 2140; 2141; 2142; 2143; 2144; 2145; 2146; 2147; 2148; 2149; 2150; 2151; 2152; 2153; 2154; 2155; 2156; 2157; 2158; 2159; 2160; 2161; 2162; 2163; 2164; 2165; 2166; 2167; 2168; 2169; 2170; 2171; 2172; 2173; 2174; 2175; 2176; 2177; 2178; 2179; 2180; 2181; 2182; 2183; 2184; 2185; 2186; 2187; 2188; 2189; 2190; 2191; 2192; 2193; 2194; 2195; 2196; 2197; 2198; 2199; 2200; 2201; 2202; 2203; 2204; 2205; 2206; 2207; 2208; 2209; 2210; 2211; 2212; 2213; 2214; 2215; 2216; 2217; 2218; 2219; 2220; 2221; 2222; 2223; 2224; 2225; 2226; 2227; 2228; 2229; 2230; 2231; 2232; 2233; 2234; 2235; 2236; 2237; 2238; 2239; 2240; 2241; 2242; 2243; 2244; 2245; 2246; 2247; 2248; 2249; 2250; 2251; 2252; 2253; 2254; 2255; 2256; 2257; 2258; 2259; 2260; 2261; 2262; 2263; 2264; 2265; 2266; 2267; 2268; 2269; 2270; 2271; 2272; 2273; 2274; 2275; 2276; 2277; 2278; 2279; 2280; 2281; 2282; 2283; 2284; 2285; 2286; 2287; 2288; 2289; 2290; 2291; 2292; 2293; 2294; 2295; 2296; 2297; 2298; 2299; 2300; 2301; 2302; 2303; 2304; 2305; 2306; 2307; 2308; 2309; 2310; 2311; 2312; 2313; 2314; 2315; 2316; 2317; 2318; 2319; 2320; 2321; 2322; 2323; 2324; 2325; 2326; 2327; 2328; 2329; 2330; 2331; 2332; 2333; 2334; 2335; 2336; 2337; 2338; 2339; 2340; 2341; 2342; 2343; 2344; 2345; 2346; 2347; 2348; 2349; 2350; 2351; 2352; 2353; 2354; 2355; 2356; 2357; 2358; 2359; 2360; 2361; 2362; 2363; 2364; 2365; 2366; 2367; 2368; 2369; 2370; 2371; 2372; 2373; 2374; 2375; 2376; 2377; 2378; 2379; 2380; 2381; 2382; 2383; 2384; 2385; 2386; 2387; 2388; 2389; 2390; 2391; 2392; 2393; 2394; 2395; 2396; 2397; 2398; 2399; 2400; 2401; 2402; 2403; 2404; 2405; 2406; 2407; 2408; 2409; 2410; 2411; 2412; 2413; 2414; 2415; 2416; 2417; 2418; 2419; 2420; 2421; 2422; 2423; 2424; 2425; 2426; 2427; 2428; 2429; 2430; 2431; 2432; 2433; 2434; 2435; 2436; 2437; 2438; 2439; 2440; 2441; 2442; 2443; 2444; 2445; 2446; 2447; 2448; 2449; 2450; 2451; 2452; 2453; 2454; 2455; 2456; 2457; 2458; 2459; 2460; 2461; 2462; 2463; 2464; 2465; 2466; 2467; 2468; 2469; 2470; 2471; 2472; 2473; 2474; 2475; 2476; 2477; 2478; 2479; 2480; 2481; 2482; 2483; 2484; 2485; 2486; 2487; 2488; 2489; 2490; 2491; 2492; 2493; 2494; 2495; 2496; 2497; 2498; 2499; 2500; 2501; 2502; 2503; 2504; 2505; 2506; 2507; 2508; 2509; 2510; 2511; 2512; 2513; 2514; 2515; 2516; 2517; 2518; 2519; 2520; 2521; 2522; 2523; 2524; 2525; 2526; 2527; 2528; 2529; 2530; 2531; 2532; 2533; 2534; 2535; 2536; 2537; 2538; 2539; 2540; 2541; 2542; 2543; 2544; 2545; 2546; 2547; 2548; 2549; 2550; 2551; 2552; 2553; 2554; 2555; 2556; 2557; 2558; 2559; 2560; 2561; 2562; 2563; 2564; 2565; 2566; 2567; 2568; 2569; 2570; 2571; 2572; 2573; 2574; 2575; 2576; 2577; 2578; 2579; 2580; 2581; 2582; 2583; 2584; 2585; 2586; 2587; 2588; 2589; 2590; 2591; 2592; 2593; 2594; 2595; 2596; 2597; 2598; 2599; 2600; 2601; 2602; 2603; 2604; 2605; 2606; 2607; 2608; 2609; 2610; 2611; 2612; 2613; 2614; 2615; 2616; 2617; 2618; 2619; 2620; 2621; 2622; 2623; 2624; 2625; 2626; 2627; 2628; 2629; 2630; 2631; 2632; 2633; 2634; 2635; 2636; 2637; 2638; 2639; 2640; 2641; 2642; 2643; 2644; 2645; 2646; 2647; 2648; 2649; 2650; 2651; 2652; 2653; 2654; 2655; 2656; 2657; 2658; 2659; 2660; 2661; 2662; 2663; 2664; 2665; 2666; 2667; 2668; 2669; 2670; 2671; 2672; 2673; 2674; 2675; 2676; 2677; 267

1. 2. 3.

NAME

sigblock, sigmask - temporarily block signals

SYNOPSIS

```
#include <signal.h>
```

```
long sigblock(long mask);  
#define sigmask(signum)
```

DESCRIPTION

sigblock is used to temporarily block the reception of signals. The input parameter **mask** is a bit vector that specifies which signals are to be blocked; a 1 in a bit *n* will block signal *n*+1. The mask is bitwise-ored with the current signal mask to create the new signal mask.

sigmask is a macro that can be used to calculate signal masks for *sigblock*. It takes a signal number (as listed in *KSIGNAL(2)*) as an argument and returns a mask.

If a signal is sent to a process but is blocked, the event is recorded for later release by *sigsetmask*. Blocked signals are not stacked; further occurrences of a blocked signal will overwrite any previous pending signal of the same *signum*.

It is not possible to block SIGKILL, SIGCONT, or SIGSTOP. This restriction is silently imposed by the system.

RETURN VALUE

The previous value of the signal mask is returned.

SEE ALSO

kill(2V), sigsetmask(2), ksignal(2)

signal, signal - temporarily block signals

at this

include <signal.h>

void sigblock(long mask);

void sigmask(int mask);

DESCRIPTION

sigblock is used to temporarily block the reception of signals. The input parameter mask is a set of bits which specifies which signals are to be blocked; a 1 in a bit n will block signal n. The mask is bitwise-ORed with the current signal mask to create the new signal mask.

sigmask is a macro that can be used to calculate signal masks for sigblock. It takes a signal number (as listed in KERNLIB(2)) as an argument and returns a mask.

When a signal is sent to a process but is blocked, the event is recorded for later release by the process. Blocked signals are not stacked; further occurrences of a blocked signal will overwrite any previous pending signal of the same signal.

It is not possible to block SIGKILL, SIGCONT, or SIGSTOP. This restriction is slightly relaxed by the system.

RETURN VALUE

The previous value of the signal mask is returned.

SEE ALSO

sigset(2), sigaction(2), signal(2)

NAME

sigsetmask - set signal mask

SYNOPSIS

```
#include <signal.h>
```

```
long sigsetmask(long mask);  
#define sigmask(signum)
```

DESCRIPTION

sigsetmask is usually used to restore signal masks after modification by *sigblock*. The parameter *mask* is the absolute value the process signal mask will be set to (compare to *sigblock*, which adds the argument to the set of blocked signals).

If there are pending instances of signals which become unblocked by the *sigsetmask* call, they are 'released': they are released into the system signal queue and their 'pending' status is cleared. [The system signal queue is maintained by the kernel null process, and is used in situations where signals could not normally be sent (such as interrupt handlers)]

sigmask is a macro that can be used to calculate signal masks for *sigsetmask*. It takes a signal number (as listed in *KSIGNAL(2)*) as an argument and returns a mask.

RETURN VALUE

The previous value of the signal mask is returned.

CAVEATS

If somehow the process re-blocks a signal released by *sigsetmask* before the system signal queue processes it, it will be blocked and marked as pending. This can happen if a signal handler makes a *sigblock* call.

SEE ALSO

kill(2V), sigblock(2), ksignal(2)

NAME

sigsetmask - set signal mask

SYNOPSIS

#include <signal.h>

sigsetmask(long mask);

sigsetmask(sigset_t mask);

DESCRIPTION

sigsetmask is usually used to restore signal masks after modification by sigblock. The parameter mask is the absolute value the process signal mask will be set to (compare to sigblock which adds the argument to the set of blocked signals).

If there are pending instances of signals which become unblocked by the sigsetmask call, they are released: they are released into the system signal queue and their 'pending' status is cleared. [The system signal queue is maintained by the kernel null process, and is used in situations where signals could not normally be sent (such as interrupt handlers).]

sigsetmask is a macro that can be used to calculate signal masks for sigsetmask. It takes a signal number (as listed in KSIGNAL(2)) as an argument and returns a mask.

RETURN VALUE

The previous value of the signal mask is returned.

CAVEATS

If somehow the process re-blocks a signal released by sigsetmask before the system signal queue processes it, it will be blocked and marked as pending. This can happen if a signal handler makes a sigblock call.

SEE ALSO

kill(2), sigblock(2), signal(2)